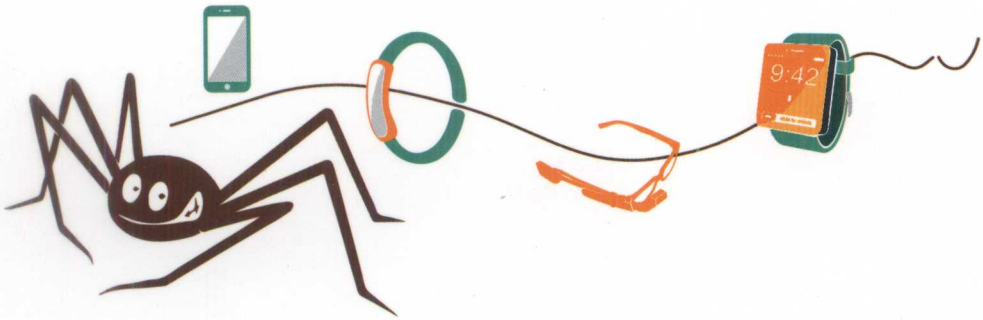
Broadview®
www.broadview.com.cn

自己动手设计物联网

黄峰达 著



中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

作者简介

黄峰达

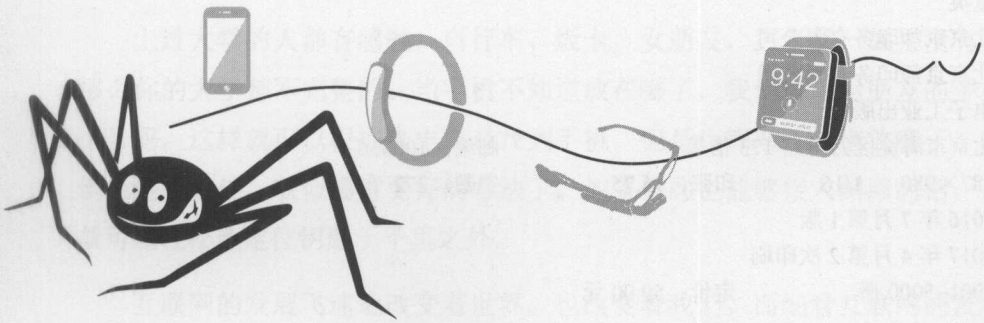
毕业于西安文理学院电子信息工程专业，现就职于ThoughtWorks。长期活跃于开源软件社区GitHub，并编写有相当数量的物联网相关开源软件、维护物联网相关资料。

专注于物联网和前端领域，长期为InfoQ编写《物联网周报》，著有电子书《一步步搭建物联网系统》、《GitHub 漫游指南》，被CSDN授予前端博客专家称号。曾作为技术专家，审阅英文版Packt出版社的物联网书籍《Learning Internet of Things》，并翻译该书。

个人网站：https://www.phodal.com/
个人微信公众号: phodal

自己动手设计物联网

黄峰达 著



電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

物联网是一个跨领域的学科，涉及方方面面的知识，包括硬件、软件、网络、协议等，这些知识很难在一本书里详细展开。目前很多关于物联网的图书集中于射频、ZigBee、WiFi、蓝牙等硬件层级的构建。本书从自己动手打造一个物联网出发，旨在教会读者如何从系统级别、架构级别去设计物联网，从而掌握打造物联网系统的全过程。

本书在讲解的过程中遵循循序渐进的思想。首先，设计一个基于文本文件的物联网系统，向读者展示一个基本的物联网体系。然后，实现以互联网为基础的物联网系统，即以 HTTP 协议与 Web 编程为基础的物联网系统。最后，打造一个能结合多个物联网协议的物联网系统。在这个过程中，读者还将学会如何打造物联网的相关应用——手机 APP、温度趋势、网页端控制等，以及如何打造智能、安全的物联网系统的相关内容。

本书适合对物联网感兴趣的创客、极客、程序员、设计师，对物联网感兴趣的大学、中学师生，以及想自己动手设计完整物联网的相关从业人员。

图书在版编目（CIP）数据

自己动手设计物联网 / 黄峰达著. —北京：电子工业出版社，2016.7
ISBN 978-7-121-29053-4

Ⅰ. ①自… Ⅱ. ①黄… Ⅲ. ①互联网络－应用②智能技术－应用 Ⅳ. ①TP393.4②TP18

中国版本图书馆 CIP 数据核字（2016）第 131901 号

责任编辑：董英
印　　刷：北京京师印务有限公司
装　　订：北京京师印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　　印张：15.75　　字数：272 千字
版　　次：2016 年 7 月第 1 版
印　　次：2017 年 4 月第 2 次印刷
印　　数：4001~5000 册　　定价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

序

很久之前读过一本关于芯片技术的书籍，在书的前言中作者讲述了一个故事。大意是这位教授在某次讲座中说到：“以后芯片的数量肯定是现在的几十倍，到时候我们都会享受到各种芯片强大的计算能力。”

这时一位听众当场起来反驳他道：“难道这栋大厦也会安装几个芯片吗？”随后引来哄堂大笑。

十年后当这位教授再次来到这栋大厦开讲座时，这里已经安装了不止几十个芯片。每个门侧都有一个门禁芯片，走廊里面各种传感芯片负责感应温度变化来检测火情等。

反观现在，如果我们说“以后电视、冰箱、空调甚至菜刀、锅碗、窗户都可以上网”，也一定会迎来不少人的嘲笑。但是我们依然有理由坚信——这些正在慢慢成为现实。

上过大学的人都有感触，自行车、饭卡、女朋友，三者其一如果没有弄丢过，那么你的大学是不完整的。当手机不知道放在哪了，我们可以用朋友的手机拨通这个号码，这样就可以根据来电声音找到手机。但是当钥匙不知道放哪了，除了翻箱倒柜地找之外，貌似没有更好的办法了。如果钥匙也能够接入网络的话，那么我们就可以轻松地定位钥匙于千里之外。

互联网的发展飞速地改变着世界，也改变着我们。而细看互联网的发展你会发现，之前我们使用互联网检索信息，这时的互联网连接的是“人与信息”，可以广义地称之为“人与物”。随着 QQ、微信、人人网等社交网站的兴起，互联网连接的是“人与人”。而下一个互联网的发展方向则是连接“物与物”——我们称之为物联网。

小时候我们总会幻想有一个哈利·波特那样的魔法世界，或者西游记那样的神话世界，而长大后我们则会畅想钢铁侠那样的科幻世界。

下班后当你拖着疲惫的身体坐在地铁上，拿出手机更新自己的状态为下班路上，这时电饭煲会自动开启，里面的米饭进入蒸煮状态，窗户自动关闭，而空调自动打开并根据室内温度调整到最适合的温度……不过最霸气的当然还是每天早上，你睁开朦胧睡眼，大喊一声“要有光”，于是整个屋子都亮了。

虽然如此便捷，但是你是一个保守派，于是你并没有把 root 权限给它们，凡是都要亲自做决定，这时打开 E-mail 将是你最忙的时候。

冰箱给你发来 2 封邮件，“牛奶不够了，我已经查询了各大商城和附近超市，建议从家乐福订购打折促销的纯牛奶”，“提示：最新发现您的各项饮食指标不平衡，所摄取的食物中维生素 C 的含量偏低，是否购买半斤芹菜。”

床发来 1 封邮件，“最近七天睡眠质量图表如下……”

就连菜刀都发来了 1 封邮件：“我该磨了。”

而这一切天方夜谭似的故事正在工程师们的智慧和努力下成为现实。本书就讲述了如何设计并构建一个完善的物联网系统。

最初在网上了解Phodal是那篇在GitHub连击300天的文章，按活跃度来讲Phodal应该是国内最活跃的开源贡献者了。

当作者 Phodal 告诉我，他的书要出版了，让我写个序。我最初以为是 Growth 增长全栈，当他告诉我是关于物联网的时候，确实有些吃惊。当我再重读 Phodal 的《编程之路》时发现，他不仅仅是一名前端加后端的全栈工程师，更是一位软件加硬件的全能工程师。

去年 Phodal 曾向我问 Flarum 接口 API 的情况，当时我正在国内为 Flarum 布道，遂给他介绍了一下 Flarum 的源码、接口、文档，以及中文社区的建设情况。没过几天 Phodal 告诉我，他用 Growth 为 Flarum 做了一个全平台的客户端——iOS、Android、

WinPhone、Windows、Mac、其他。但愿有一天我家的微波炉也可以在 Flarum 论坛发帖。

Phodal 来写这本物联网的书，那么注定这本书将是一本科普与实战相结合的进阶读物。里面不会有太多的名词概念的解释，更多的是如何设计，如何实现。

如果你是一名创客、极客，那么这本书很适合你。

Just enjoy it.

justjavac

Flarum 中国社区创始人

2016 年 5 月，天津

前　言

在最开始的时候，人们使用 CGI 来开发动态网页。在那个时候，人们也使用 CGI 来开发一些联网应用。早先及现有的物联网应用使用 Web 技术作为基础的技术，接着人们开始将一些不错的协议使用在物联网中，如 MQTT。又有一些组织在制定一些协议，如 CoAP。同时越来越多的国家和组织在出台自己的标准。

幸运的是，这些技术都是依据现有的技术发展而来的。技术在过去的几十年里发生了快速的变化，但是它们的本质是解决问题。而问题并没有发生快速的变化——还是一样的问题，只是我们有了更多可用的解决方案。

本书目标

本书不是一本详细介绍物联网系统的书籍，也不会和国内的一些教程一样主要集中于射频、ZigBee、WiFi、蓝牙等。我相信这些知识你已经在其他书中学到过了，而这些书籍更多的是侧重于硬件层级的构建，没有从系统级别、架构级别对系统进行设计。设计这样一个完整的系统，则是本书的核心。本书旨在教会用户如何去设计的思想，以及如何打造物联网系统的过程。至于系统底层硬件的实现细节则需要用户去把握。

本书遵循循序渐进的思想，从设计一个极其简单的物联网系统，再到基于以互联网为基础的物联网系统，最后打造了一个基于物联网协议的物联网系统。在这个过程中我们还将教会读者打造物联网的相关应用——手机 APP、温度趋势、网页端控制等。

由于设计这个物联网系统本身有太多的知识点，并且涉及方方面面的知识，本

书将尽可能地向读者推荐一些扩展阅读资料，并且建议读者多多实践。同时，由于章节间是一步步加深的关系，如果你在这其中遇到什么问题可以及时与作者联系。

目标读者：对物联网感兴趣的创客、极客、程序员、设计师；对物联网感兴趣的大学、中学师生；想自己动手设计完整物联网的相关从业人员。

为什么是 JavaScript

在思考着用哪门语言来编写程序的时候，我考虑到了 Python、Java、JavaScript。Python 是我最喜欢的语言，JavaScript 是我最擅长的语言，Java 是我最常用（工作）的语言。它们都是非常不错的跨平台语言，它们都有广泛的使用者。

如果考虑将其商业化，我会考虑使用 Java 语言。Java 语言是一门“正统”的语言，即在国内的计算机科学领域，各个院校都将之列为必学语言。除去 Java Web 的流行带来的需求，Android 也增大了对 Java 语言的需求。由于 Java 语言是一门编译语言，并且经过二十多年的发展，积累了大量的技术和智慧，使得它相当稳定。因而多数企业都采用这门语言作为其主要语言。尽管 Java 语法简单，但是语法却比较烦琐、开发效率低，并不利于我们表述。

如果仅仅是考虑学习服务端，我会考虑使用 Python 来写我们的物联网系统。Python 是一门简单、易学、易懂的脚本语言，在科学计算领域非常受欢迎。并且，在书中的一些例子里我会使用 Python 语言。Python 语言富有表现力，可以更容易地让我们将自然语言转换为机器语言。然而，对于读者来说可能存在更多的学习成本。

So，JavaScript 有什么优点？无论我们使用 Java 还是 Python 语言来开发我们的物联网应用，凡是涉及到网页前端相关的内容，我们都需要 JavaScript，这就足够了！

那么，为什么我们不使用 JavaScript 来完成所有的这些工作呢？它有：

- Node.js 框架——最流行的 JavaScript 服务端平台，可以创建 Web 应用。
- Cordova 框架——最流行的混合应用框架，可以使用 Web 技术来开发手机应用（iOS、Android、Windows Phone 等）。

这就意味着，我们可以用这门语言完成所有的开发任务——服务端、客户端、移动应用。如果你想，也可以用这门语言完成硬件端的开发。三星推出了可以适用于嵌入式设备的小型 JerryScript，谷歌的两名前员工推出了适用于物联网领域的软件平台 Smart.js。并且已经有几个开发板，如 Tessel 2、Espruino，可以让你使用 JavaScript 为你的芯片编程。

其他语言

在服务端、客户端和移动应用的例子里，我们会用 JavaScript、HTML、CSS 来向读者展示其中的原理。在硬件部分，我们会用 C/C++语言。在 Raspberry Pi 上编程的时候，会使用 Python。其他部分，如 Dashborad 和 NodeMCU 的介绍，会有一些 Lua 或者 Ruby 语言的例子。

我们之所以在 Raspberry Pi 上使用 Python 语言，不仅仅是因为使用 Python 语言更容易读懂，而且在这个领域中主要使用的也是 Python 语言。同理于 Dashboard 的示例，我们使用了 Dashing 这个流行的框架，只是因为它更容易上手，并且使用的人较多。这就意味着，在我们遇到问题的时候更容易解决。

我们并不希望这些语言会阻碍你前进。在适当的时候你可以先跳过这一部分——如 Dashing，我们展示了如何自己去写类似的界面，使用这个框架只是为了开发更快。随后在真正使用它的时候去深入它们。

为了用而学习是最有效率的学习。

本书内容

如下所示，本书分为 8 个章节和 4 个附录。

第 1 章　概览

本章介绍了物联网的历史背景和相关技术，以及其与互联网的关系。

第 2 章　一个极简的物联网：hello,world

本章以一个文本文件的数据为中心，快速搭建一个极简的物联网原型。其中将向读者展示物联网的一些基础知识。

第 3 章　分解物联网系统

本章将描述常用的物联网系统架构，并对每一个层级进行详细的描述和介绍。

第 4 章　基于 Web 的物联网系统

本章将介绍常用 API 的模式 RESTful，并将带领读者打造基于 HTTP 协议的物联网系统。

第 5 章　连接设备

本章将介绍一些容易上手的设备，并将这些设备连接到物联网。不仅可以控制 LED，也可以上传传感器数据。

第 6 章　物联网应用示例

本章将介绍用趋势图来显示传感器的数据，还将展示如何打造一个跨平台的 APP 来控制物联网设备。同时，还有使用一些当前的云服务来降低开发难度。

第 7 章　实现超越互联网的物联网

本章将介绍 MQTT、CoAP 等物联网协议，它们可以帮助我们更好地处理物联网系统中的消息通信。

第 8 章　智能与安全

本章将关注于一些额外的话题，如安全、智能、私有化。

附录

附录提供了简单的 JavaScript 入门指引、Ionic 的一些介绍、物联网资料及

Raspberry Pi 的初始化等。

代码

本书的代码都可以从 Github 上（https://github.com/phodal/iot-code）或者 CSDN Code 上（https://code.csdn.net/designiot/code）下载到。

代码以 MIT 协议公开，你可以将其中的代码用于你的开发或者项目中。如果你在这个过程中遇到一些问题，**请在网页上创建一个相关的 Issues**，以便我们能收到这个问题，并能**帮助其他遇到同样问题的人解决问题**。

下面是代码的一些简介，你也可以在相关的网页上看到。

目录	功能
手机 APP	目录下是 Hybird 应用相关代码
仪表盘	仪表盘相关代码
第 2 章	Nginx 示例、Python 基础
第 4 章	基础 Web 服务器代码、RESTful 服务代码
第 5 章	硬件、传感器示例、RESTful 服务代码
第 6 章	RESTful 服务器代码、Dashboard 代码
第 7 章	MQTT、CoAP 协议的服务器代码
第 8 章	自然语言处理示例

希望读者没有被上面的知识点所困扰到，知识点越多就说明越有挑战性！难道不是吗？

在线资源

由于笔者本身是开源的重度参与者及物联网的爱好者，并且这些资源可以时时更新。下面是一些在我的 GitHub 上关于物联网的相关资源。

1. https://github.com/phodal/awesome-iot 项目收集了各式各样的物联网资料——如框架、库、操作系统、API、平台、硬件等，大约每周会更新一次。
2. https://github.com/phodal/designiot 项目是笔者之前写的《一步步搭建物联网系统》，也是这本书的前身。
3. https://github.com/phodal/lan 项目是本书的物联网系统中的原型，但是含有更多的知识点和模块。
4. https://github.com/phodal/iot 项目是一个基于 PHP 语言的 Laravel 框架的最小物联网系统。
5. https://github.com/phodal/iot-document 项目内容已经收录到了本书的附录中。
6. https://github.com/phodal/designiot-refs 项目将包含本书的一些扩展阅读资料。
7. https://github.com/phodal/designiot-images 项目包含了本书的所有图片——由于打印出来的是黑白版的，可能会影响阅读。
8. https://github.com/phodal/growth 项目则包含了读者需要的 Web 开发的相关资料。

遇到问题

在阅读本书的过程中，如果读者遇到一些问题，可以通过以下方式与我联系：

1. 邮箱: h@phodal.com
2. 微博: @phodal
3. QQ 群: 348100589
4. 微信公众号: Phodal
5. GitHub: @phodal（读者如果在相关代码上有问题，可以直接使用 GitHub 的 Issue 来提问）。
6. 论坛: http://bbs.designiot.cn/

目　录

第 1 章

概览

本章内容

- 学习物联网的概念及其背景
- 物联网涉及的技术
- 物联网的未来

在我初识物联网这个词时，我在网上、书上等资料上看到的很多字眼都是 RFID **无线射频识别**，随后又看到了传感器网络等概念。这些词并没有告诉我们物联网是什么，而且它也有些以偏概全。我们所说的物联网来源于英语“Internet of Things”，大意即事物的网络。这里就有疑惑了，什么才算得上这里的事物呢？我们日常用的纸、笔也算是事物，但是它似乎不属于这里的物——因为我们没法拿它连接上网络。

1.1 物联网发展历史

讲物联网之前，我们也需要好好地介绍一下互联网。互联网诞生于 20 世纪 70 年代，是由美国国防部出于军事目的开始制定的。当时这个名为 ARPA 网的网络只连接了四台主机。

随后在 1974 年，ARPA 的罗伯特·卡恩和斯坦福的温登·泽夫提出 TCP/IP 协议。而到了 1983 年 1 月 1 日，ARPA 网才将其网络核心协议由 NCP 改变为 TCP/IP 协

议。1986 年，美国国家科学基金会创建了大学之间互联的骨干网络 NSFnet，这使互联网走向了世界。紧接着在 1991 年 8 月，Tim Berners 和其他在欧洲粒子物理实验室的人创建了世上第一个网站。

互联网从军用领域走向公用领域花了几十年的时间，而物联网的发展则更为快速。

通常来说，人们所说的物联网（The Internet of Things）的概念最早于 1999 年由美国 Auto-ID 公司提出，当时主要是建立在物品编码、RFID 技术和互联网基础上。这也解释了为什么我们看到的很多物联网相关教材多数都在讲 NFC、RFID 技术。与此同时，中科院在当时开始了传感网的研究。传感网顾名思义是传感器组成的网络——更多的是集中于收集传感器的数据，算得上是物联网的一个子集。

在更早之前，1995 年比尔·盖茨的《未来之路》一书也提出了物联网的概念。只是由于技术水平、硬件等的限制，在当时并没有引起重视。

随后 2005 年，国际电信联盟发布了《ITU 互联网报告 2005：物联网》，报告中指出：无所不在的“物联网”通信时代即将来临。这时人们对于物联网已经有了新的认识——不再仅仅局限于 RFID 技术。只是由于当时技术的限制及应用的限定，并没有产生特别大的影响。

在 2009 年 1 月，IBM 首席执行官彭明盛提出“智慧地球”构想，其中物联网为“智慧地球”不可或缺的一部分。同年，奥巴马在就职演讲后已对“智慧地球”构想提出积极回应。温家宝总理在 8 月提出了“感知中国”，物联网被正式列为国家五大新兴战略性产业之一。

根据美国权威咨询机构 FORRESTER 预测，到 2020 年，世界上物物互联的业务，跟人与人通信的业务相比，将达到 30∶1。

不过在今天，物联网已经使生活产生了翻来覆去的变化。开发人员现在也是尽他们所能去连接一切能连接的物。

1.2 物联网概念

在几十年前，能连接上网络的主要设备就是计算机。对于计算机来说，这个网络就是 Internet，即互联网。而在十几年前，能连接上网络的设备就又多了一个手机——这完全改变了人类今天的生活。十几年前物联网也诞生了，只是当时的物联网是基于 RFID 技术的。

由于十几年前对于物联网的定义是 RFID 技术，这也使得最近几年来的一些物联网书籍有一些不足。它们将主要精力放置于射频、Zigbee、蓝牙等一些通信技术上，而这些技术多数则是用于区域内的设备通信的。换句来说，只集中于内部通信，在连接到网络的时候就会遇到种种困难。而这也是物联网需要解决的主要问题，连接到网络。

物联网来源于 Internet of Things 一词，即世间万物的互联网。顾名思义，物联网的意思就是物物相连的互联网。这有两层意思：第一，物联网是建立在互联网之上的，是互联网的拓展和延伸；第二，其用户端扩展和延伸到了物品与物品之间，进行信息通信和交换。

物联网有如下特征：

- 首先，广泛应用了各种感知技术。在物联网中部署了大量的多种传感器，每个传感器都能从外界采集信息，不同类的传感器捕获的信息不同。而且获得的数据具有实时性，按照一定的规律来采集数据，不断更新数据。
- 其次，它是建立在互联网上的网络。物联网技术的核心和基础仍是互联网，通过各种无线和有线网络与互联网结合起来，将物体的信息准确实时地传递出去，数据传输过程中必须适应各种网络协议。
- 还有，物联网本身也具有一种智能处理的能力，能够智能控制物体。物联网从传感器中获得数据，然后进行分析，处理有意义的数据，来适应不同用户的需求。

当前物联网已经在多方面开始应用，如远程抄表、电力行业、视频监控等。以

及在物流领域和医疗领域也都日趋成熟，如物品存储及运输监测、远程医疗、个人的健康监护等。除此之外在环境监控、楼宇节能、食品等方面也开展了广泛应用。

1.3 联网——各式各样的联网设备

物联网的定义有些广泛，手机、计算机等都属于事物。互联网和物联网之间有相当多的交集。如我们在很多地方看到的一样，物联网的核心和基础仍将是互联网。而这仅仅只是在当前的环境下产生的一些技术。在可预见的未来，我们将会看到物联网和互联网的越来越多的不同之处。不同的环境、条件下，我们可能会使用不同于互联网的架构模型，但是其在大体上也是一致的。

由于上网的设备不再仅仅局限于手机、电脑、平板，这也意味着越来越多的联网设备将被开发出来。在当前我们也可以通过现有的联网设备——手机、电脑等，将我们的物连接到互联网中，使之成为互联网中的物联网。

针对于不同的环境，我们就可能会组成由不同的设备构成的联网系统。

设备本身的一些特有属性的关系，限制了它们的连接方式，如：

- 当我们使用红外线遥控的时候，我们就需要使用红外线作为传输媒介。
- 当我们使用手机来控制玩具的时候，我们就需要用蓝牙作为传输媒介。
- 当我们需要在电脑和手机之间传输资料的时候，我们就需要数据线或者 WiFi 作为传输媒介。

使用特定的传输媒介，也会决定我们将使用不同的协议：如蓝牙协议、红外通信协议、NFC 等。而这些不同的设备也需要借助于其他联网设备，来向服务中心通信。

对于这些联网设备来说，它们可能会使用不同的传输协议：

- HTTP，即超文本传输协议，它是当今互联网的基础协议。
- CoAP，受限制的应用协议，它面向那些资源受限制的物联网设备。

- MQTT，即消息队列遥测传输，它被设计用于轻量级的发布/订阅式消息传输。
- ……

这些不同的协议、不同的设备连接到网络，也决定了整个系统看起来相当复杂。如图 1-1 所示的系统将是现在到未来几年内，家居联网系统的一个简化。

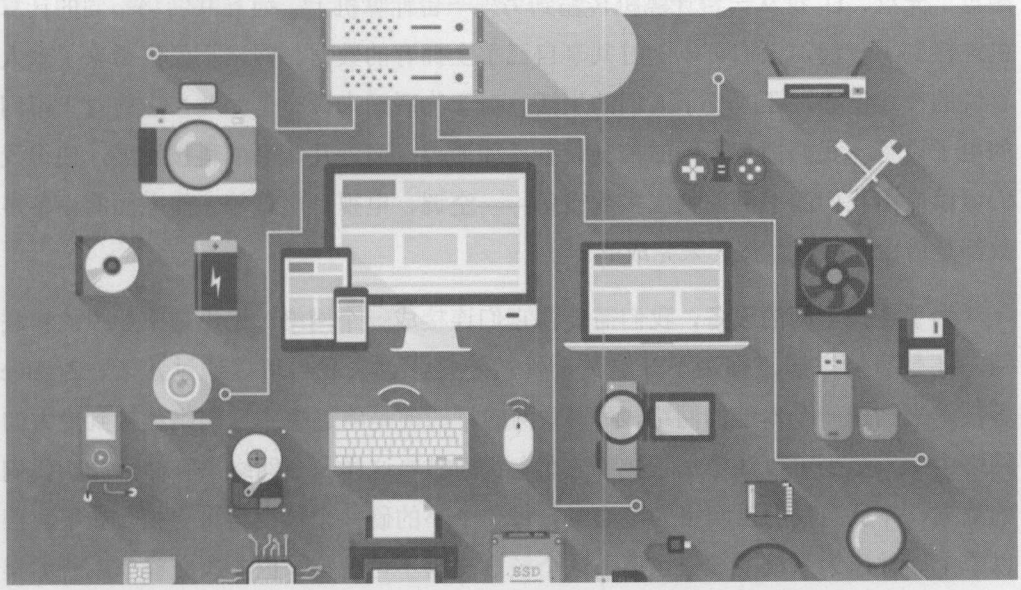


图 1-1　物联网设备集

现在我们已经连接了不同的设备——手机、电脑、平板、摄像头、打印机，我们还可以连接上洗衣机、电饭锅、电暖器、空调等，我想不到未来有什么用电的设备不会连接到网络。

1.4　物联网系统的核心是网络

过去我也花了大量的时间在硬件的软件开发上，硬件上的软件和服务端的软件开发有很多不同之处，最显著的特征是：服务端的软件开发是持续不断的。当我们这个版本里引入了一个 Bug，那么我们可以快速地修复这个 Bug。而硬件上的软件

开发的产出则是固件，通俗地来说就是固定的软件，这是一个很大的转变。同样，对于物联网系统设计来说也是如此。过去人们将主要的精力集中于硬件端的设计，而不是深入物联网的核心部分网络连接。

在互联网普及之前，人们使用自己的 PC 在上面记录自己的数据，数据只能通过磁盘、光盘、U 盘从一台计算机传输到另外一台计算机上。而互联网将不同的计算机连接到了一起，人们可以随时共享自己手上拥有的数据。网络不仅仅带来了令人喜悦的在线世界，还改变了人们的生活。对于物联网来说亦是如此，之前对于不同的电子设备，我们拥有不同的控制设备，如遥控。而近几年来的科技发展，使得我们可以借由我们的手机控制大多数设备——空调、电视、电灯等。而上面的设备多数都是借由互联网、蓝牙来完成的。

为了连接不同的设备，我们需要将它们连接成一个网络。无论是工业中的物联网，还是智能家居，都离不开这关键性的连接。过去我们可以通过蓝牙、ZigBee 等技术来构建一个本地的网络，而物联网的目的便是将这个本地网络连接到更大的网络中——至少可以让你远程控制它们。我们需要这样一个网络来存储并分析我们的数据，并且当我们以网络为核心时，每一个小的硬件个体变得可更换。如果我们以单个硬件作为系统的核心，那么这个核心变得不可替换，也会改变整个系统的架构。而这样的硬件在未来可以作为协调这些设备的中心。同时，当这个硬件将要被淘汰时，我们也可以使用最新的技术来开发我们的系统。

因而你可以预见到，这是一个很复杂的领域。如果你是一个硬件开发人员，这里就需要结合你之前学习到的单片机的知识，同时你可能需要去了解 Web 开发的知识。作为一个软件开发人员，你需要去了解更多的物联网相关协议，以及硬件相关的通信知识。

1.5 小结

本章简单地对物联网的过去做了一些介绍，并且在可预见的未来它是一个非常重要的领域。而在这个领域中，涉及硬件、软件、Web 开发等各种知识，这使得系

统变得尤为复杂。并且由于一些硬件设备的使用条件限制，使得连接不同的设备变得很复杂——我们需要处理不同的通信设备、协议，以及不同的网络环境。而正是这些复杂的设备环境，让我们可以认识到其实摆在我们眼前最难的问题是：把它们都连接到网络。

下一章，让我们先做一个简单的网络以供它们连接吧。

第2章

一个极简的物联网：hello,world

本章内容

- 数据是以怎样的形式在 Internet 中传输的
- HTTP 协议的一些相关知识
- 对控制硬件有一个基本的了解

起先，互联网是为了在不同计算机用户和通信网络之间进行常规的通信而开发的。而这里的通信，则是以互联网为媒介进行的信息交流与传递。因而，我们使用互联网的实质就是进行信息交流与传递。而信息实则加工处理后的有用数据。

什么是有用的数据？有用是相对的，我们将一些饮料瓶当成了垃圾，但是在那些饮料厂看来，那就是有用的资源。我们身处在 IT 这个行业中，除非我们所处的业务、领域与生物有关，否则类似于诺贝尔生物学奖这样的新闻，看上去便是无用的“垃圾”。

今天，互联网也是以数据为中心进行传递的，物联网也是如此。两者稍有不同的是，后者更多的是将时间花费在处理数据上。因此，首先了解数据如何传输，然后才能对其进行处理。

在这一章中我们将理解如何以一个文本文件的数据为中心，快速搭建一个极简的物联网原型。开始阅读之前，先让我们看看这一章的架构图，如图 2-1 所示。

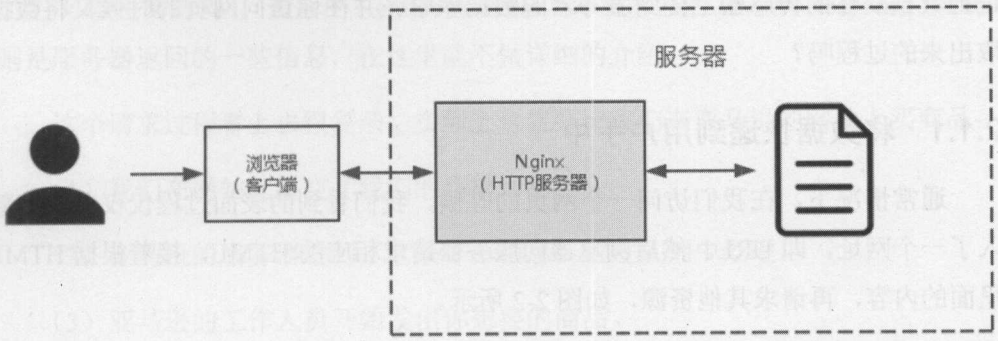


图 2-1 第 2 章程序架构图

这张图是这一章的核心，对于懂或者不懂两个极端的读者来说，这张图很简单。即一个用户的请求发送给服务器，最后由服务器返回一个文本文件中的内容。

本章所需的软件清单：

- Mongoose 是一个非常小巧的 Web 服务器，我们不需要编写任何代码便可以运行服务器。
- Nginx 是一款面向性能设计的 HTTP 服务器，也是主流的 HTTP 服务器之一。
- Curl 是一个在命令行方式下工作的开源文件传输工具。

本章所需的硬件清单：

- Raspberry Pi 或同等的带 Linux 操作系统的硬件。

2.1 数据的传输过程

或许你会问互联网的数据是从哪里产生的？这是一个有趣的话题，但不是本章讨论的核心。

不过，我相信你已经知道了这个数据应该是由传感器产生的。而且，我相信在

你的手上已经有大量的传感器数据。也许你还使用过一些 Web 服务，并且将你的数据上传到某个 API 上；或许还在网页上打造你想要的温度趋势图。但是，你真的知道把数据从你的传感器上传到服务器的数据库中，并在你访问网页的时候又将数据取出来的过程吗？

2.1.1 将数据快递到用户手中

通常情况下，在我们访问一个网页的时候，我们看到的表面过程仅仅是我们敲入了一个网址，即 URL。然后浏览器向服务器请求相应的 HTML，接着根据 HTML 里面的内容，再请求其他资源，如图 2-2 所示。

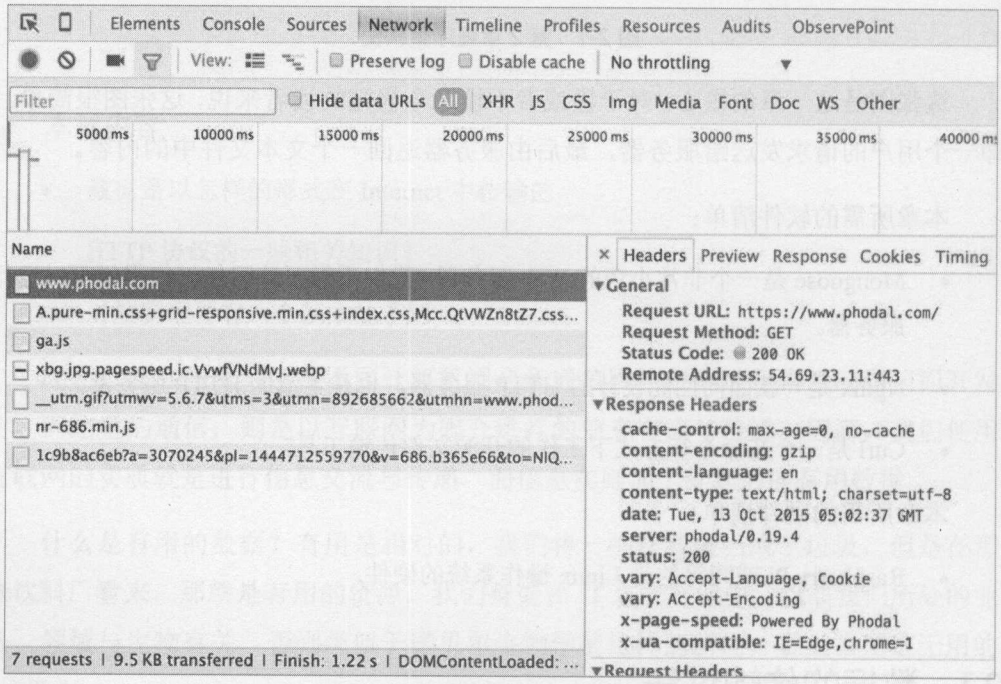


图 2-2 网站请求资源的网络过程图

在这里，我们输入了笔者的网站地址到浏览器的地址栏中：https://www.phodal.com，接着在 Chrome 浏览器（即谷歌浏览器）的 NetWork 中可以看到这个过程（在其他浏览器中也可以看到相似的内容）。在图 2-2 的左侧，我们可以看到我们

请求的所有资源。在图 2-2 右侧的 General 部分，我们可以看到我们请求的网址是 https://www. phodal.com，方法是 GET，返回的 HTTP 状态是 200（即 OK，代表这个 URL 可以正常访问），最后的 Remote Address，即服务器的 IP。而 Response Headers 则是服务器返回的一些信息，在这里就不做详细的介绍。

这个请求过程看上去很复杂，实际上这就好比我们在亚马逊的网站上买商品：

（1）我们在网站上提交了我们的订单信息。

（2）亚马逊的服务器看到了这条订单信息，放进队列中处理。

（3）亚马逊的工作人员开始发出你想要的商品。

而真实的请求过程和这个类似，我们发出了我们的请求，最后我们得到了想要的内容，如图 2-3 所示。

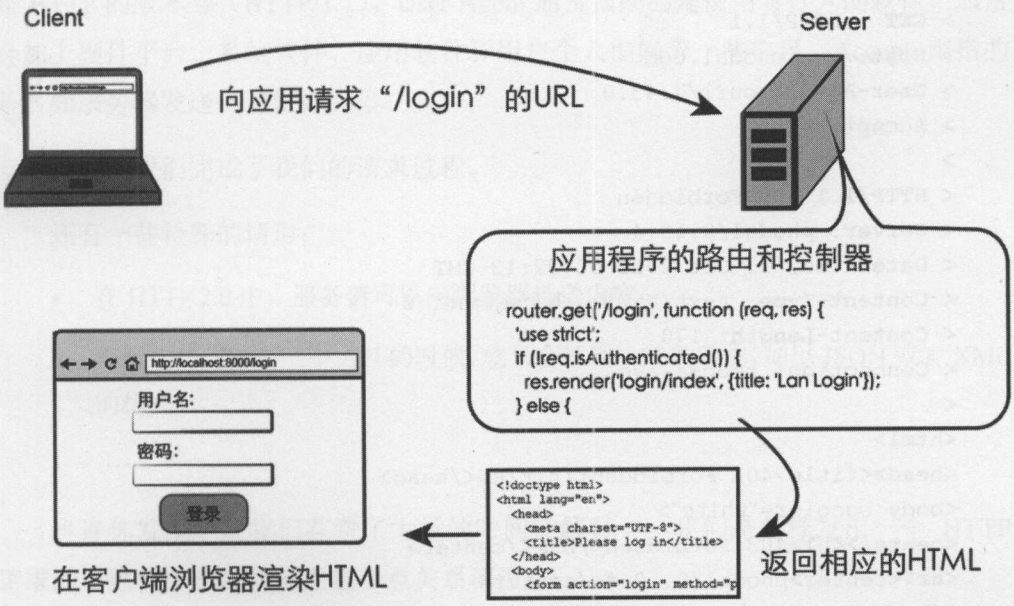


图 2-3　请求过程

但是这个过程可能会稍微复杂一点，如果你的操作系统带有 cURL[1]这个软件（GNU/Linux、Mac OS 都自带这个工具，Windows 用户可以从 http://curl.haxx.se/download.html 下载到），那么我们可以直接用下面的命令来看这个过程[2]（-v 参数可以显示一次 HTTP 通信的整个过程）：

```
curl -v https://www.phodal.com
```

我们会看到下面的响应过程：

```
* Rebuilt URL to: https://www.phodal.com/
*   Trying 54.69.23.11...
* Connected to www.phodal.com (54.69.23.11) port 443 (#0)
* TLS 1.2 connection using TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384
* Server certificate: www.phodal.com
* Server certificate: COMODO RSA Domain Validation Secure Server CA
* Server certificate: COMODO RSA Certification Authority
* Server certificate: AddTrust External CA Root
> GET / HTTP/1.1
> Host: www.phodal.com
> User-Agent: curl/7.43.0
> Accept: */*
>
< HTTP/1.1 403 Forbidden
< Server: phodal/0.19.4
< Date: Tue, 13 Oct 2015 05:32:13 GMT
< Content-Type: text/html; charset=utf-8
< Content-Length: 170
< Connection: keep-alive
<
<html>
<head><title>403 Forbidden</title></head>
<body bgcolor="white">
<center><h1>403 Forbidden</h1></center>
<hr><center>phodal/0.19.4</center>
</body>
</html>
```

1 cURL 是一个利用 URL 规则在命令行下工作的文件传输工具。

2 如果该网站支持 HTTP 2.0，那么你可以尝试用 **curl -v --http2 https://www.phodal.com**，这取决于你的 cURL 版本是否支持 HTTP 2.0。

```
* Connection #0 to host www.phodal.com left intact
```

我们尝试用 cURL 去访问我的网站，会根据访问的域名找出其 IP，通常这个映射关系来源于 ISP 缓存 DNS（英语：Domain Name System）服务器。

以“*”开始的前 8 行是一些连接相关的信息，称为**响应首部**。我们向域名 https://www.phodal.com/发出了请求，接着 DNS 服务器告诉了我们网站服务器的 IP，即 54.69.23.11。出于安全考虑，在这里我们以 HTTPS 协议为例，所以此处连接的端口是 443。因为使用的是 HTTPS 协议，所以在这里会试图去获取服务器证书，接着获取到了域名相关的证书信息。

随后以“>”开始的内容，便是向 Web 服务器发送请求。Host 即我们要访问的主机的域名，GET / 则代表着我们要访问的是根目录，如果我们要访问 https://www.phodal.com/ about/页面，在这里，便是 GET 资源文件/about。紧随其后的是 HTTP 的版本号（HTTP/1.1）。User-Agent 通常指向的是使用者行为的软件，通常会加上硬件平台、系统软件、应用软件和用户个人偏好等一些信息。Accept 则指的是告知服务器发送何种媒体类型。

就这样我们完成了我们的请求过程。

还有一些特殊的情形：

- 在 HTTP 2.0 中，服务器可以向浏览器推送内容。
- 当我们访问的是一个 API 的时候，就只会返回 API 的数据（如以 JSON 或者 XML 的格式）。
- ……

也许你发现了，我们花费了大量的时间在解释——我们是怎样发出一个 HTTP 请求的，这似乎与我们的主题一点关系都没有。

而这正是我们一直忽略的内容，上面说到的是 GET 请求，即用于获取数据的请求。这也是在浏览器开始获取页面内容之前发生的事，随后的事情你可能已经猜到了。返回器会访问一个状态，而浏览器会去解析，在上面的内容中，它以“>”开头。

随后的<html>至</html>里面的内容，便是页面的内容，最后会交给浏览器去解析。

自始至终，我们只做了一件事情，即将数据像快递一样递到用户眼前。

2.1.2 数据与服务中心

上面的传递过程只说明了数据是如何传递到用户手中的。如果单纯作为一个用户，我们了解这些已经足够了。但是作为一个开发者，我们需要知道这个订单到底在服务中心发生了什么事？

如果再仔细回溯一下上面的过程，你会发现似乎忽略掉了一些细节。实际上，在我们将数据取出来的过程中，还需要经历路由器、各个地域的 ISP 服务器等，最后才到我们的服务器。我们的服务器在某种程度上相当于一个服务中心。但是在大多数情况下，一个请求会经过多个服务器，才会传到我们的服务器中。

通常来说在网站后台，我们的请求会先交由 Web 服务器来处理。

（1）这个 Web 服务器会使用 Socket 监听端口（通常这个端口是 80，用于 HTTP 超文本传输服务。在上面的例子中这个端口是 443， 用于 HTTPS 加密的超文本传输服务）。

（2）接着会解析 header 中请求的网站的地址。

（3）再根据请求的地址做出响应——如直接返回 HTML 文件，或转给 **Web 应用服务器**（如 Java 的 Tomcat、Jetty）处理，或转给 CGI。

如上所述的 Web 服务器架构如图 2-4 所示。

在那些流量比较大的网站上，通常会有一个缓存服务器，用于将一些频繁访问的 Web 页面保存在系统中。在这种情况下，不会直接请求到 Web 应用服务器，能减少系统的负担，同时可以加快访问速度。其架构如图 2-5 所示。

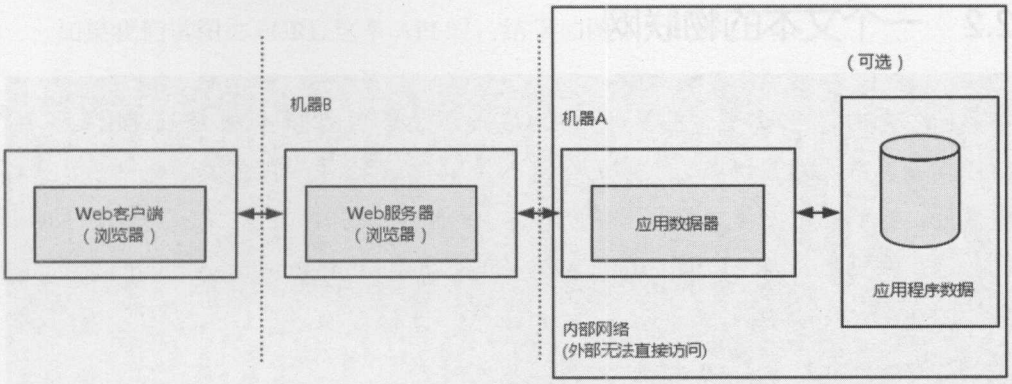


图 2-4　Web 服务器架构

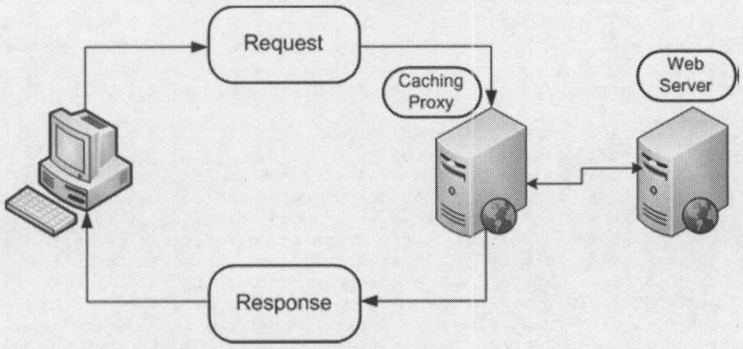


图 2-5　含缓存的 Web 服务器架构

Web 服务器不仅实现了 HTTP 和相关的 TCP 连接处理，而且还实现了 HTTP 协议、管理 Web 资源、并负责提供 Web 服务器的管理功能。常见的 Web 服务器有 Nginx、Apache、IIS，在嵌入式系统领域有 lighttpd、thttpd、minihttpd 等。像淘宝用的就是 Nginx 服务器，CSDN 用的是基于 Nginx 的扩展服务器 OpenResty，因此在接下来的章节中，我们会以 Nginx 为例来设计物联网系统。

依据我们选择的后台语言，我们需要不同的 Web 应用服务器。如在 Java 语言中，有 Tomcat、Jetty 等，在 Python 语言中，有 uWSGI、Gunicorn 等。在这个时候，Web 应用服务器会到数据库中获取相应的数据。随后，返回相应的 HTML 内容，这部分内容随后返回到 Web 服务器中。最后，返回到浏览器中渲染。

接着，我们以一个文本文件为例来搭建一个简单的物联网系统。

2.2 一个文本的物联网

在今天，无论多么复杂的 Web 应用，最后浏览器显示的时候都是 HTML 与 CSS[3]。而像我们日常说到的 API[4]，则会以 JSON、XML 等近乎人类可读的数据格式来存储数据。当我们试图在浏览器上访问一个 JSON API 的时候，浏览器会以文本的形式显示在网页上。当我访问 Github API（https://api.github.com/users/phodal）时，会有如图 2-6 所示的结果。

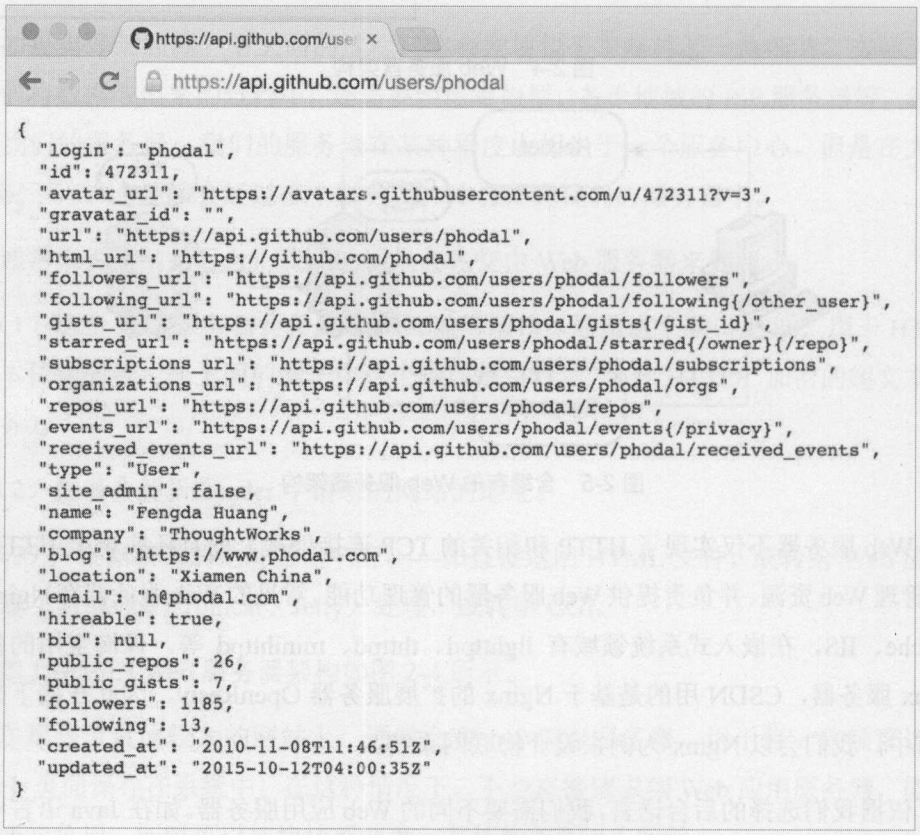


图 2-6 访问 Github API 返回的结果

3 层叠样式表（英语：Cascading Style Sheets）是一种用于为结构化文档（如 HTML 文档或 XML 应用）添加样式（字体、间距和颜色等）的计算机语言。

4 应用程序接口（英语：Application Programming Interface），是软件系统不同组成部分衔接的约定。

如果我们试图去 cURL 这个 API 时，结果如图 2-7 所示。

```
< X-Content-Type-Options: nosniff
< Vary: Accept-Encoding
< X-Served-By: 318e55760cf7cdb40e61175a4d36cd32
<
{
  "login": "phodal",
  "id": 472311,
  "avatar_url": "https://avatars.githubusercontent.com/u/472311?v=3",
  "gravatar_id": "",
  "url": "https://api.github.com/users/phodal",
  "html_url": "https://github.com/phodal",
  "followers_url": "https://api.github.com/users/phodal/followers",
  "following_url": "https://api.github.com/users/phodal/following{/other_user}",
  "gists_url": "https://api.github.com/users/phodal/gists{/gist_id}",
  "starred_url": "https://api.github.com/users/phodal/starred{/owner}{/repo}",
  "subscriptions_url": "https://api.github.com/users/phodal/subscriptions",
  "organizations_url": "https://api.github.com/users/phodal/orgs",
  "repos_url": "https://api.github.com/users/phodal/repos",
  "events_url": "https://api.github.com/users/phodal/events{/privacy}",
  "received_events_url": "https://api.github.com/users/phodal/received_events",
  "type": "User",
  "site_admin": false,
  "name": "Fengda Huang",
  "company": "ThoughtWorks",
  "blog": "https://www.phodal.com",
  "location": "Xiamen China",
  "email": "h@phodal.com",
  "hireable": true,
  "bio": null,
  "public_repos": 26,
  "public_gists": 7,
  "followers": 1185,
  "following": 13,
  "created_at": "2010-11-08T11:46:51Z",
  "updated_at": "2015-10-12T04:00:35Z"
}
* Connection #0 to host api.github.com left intact
```

图 2-7　用 cUR 获取 Github API 的结果

即和前面一样的结果，这就意味着，我们使用一个以.json 为扩展名的文件作为一个 API。

2.2.1　从浏览器到服务器

现在我们可以在本地搭建一个 Web 服务器，这里的本地源自英语 Local。换句话说，我们可以在自己的电脑上搭建这样一个 Web 服务器，也就是将我们的电脑当成

一个服务器。普通的服务器与一般计算机的架构类似，都是由处理器、硬盘、内存等构成的，只不过其中的一些元件，如处理器，可能会换成处理能力更强劲、更稳定的处理器。对于我们开发应用来说，本地的处理能力绰绰有余。

在这里，我们以 Nginx 为例，来搭建一个 Web 服务器。尽管它称为 Web 服务器，实际上在这里只是一个为用户提供服务的计算机软件。通常我们也会将运行着 Web 服务器软件的服务器（计算机）称为 Web 服务器，为了避免混淆，我们只称其为服务器。在这里只是简单地将 Nginx 作为一个静态文件服务器，当我们访问 http://127.0.0.1/api.json 时，返回的是 api.json 文件。

首先，我们需要安装 Nginx。

如果你用的是 Windows 系统，并装有 Chocolatey[5]（如果没有也可以尝试安装一下），可以在**命令提示符**下执行（需要以管理员权限执行）：

```
choco install nginx
```

安装完成后，结果如图 2-8 所示。

如果你不打算使用 Chocolatey 来完成，可以到官网下载：

```
http://nginx.org/en/ download.html
```

GNU/Linux 系统用户，如 Ubuntu、Debian，可以在**命令行**下执行：

```
sudo apt-get install nginx
```

如果是 CentOS，应该是：

```
sudo yum install nginx
```

Mac OS 系统用户，可以在**终端**下用 Homebrew[6]安装：

```
brew install nginx
```

5 Chocolatey 是 Windows 下一款命令行包管理软件。安装方法可以参见官网 https://chocolatey.org/，或者在命令行执行 @powershell -NoProfile -ExecutionPolicy Bypass -Command "iex ((new-object net.webclient).DownloadString('https:// chocolatey.org/install.ps1'))" && SET PATH=%PATH%;%ALLUSERSPROFILE%。

6 Homebrew 是一款自由及开放源代码的软件包管理系统，用以简化 Mac OS X 系统上的软件安装过程。

```
Extracting  nginx-1.6.2\docs\CHANGES
Extracting  nginx-1.6.2\docs\CHANGES.ru
Extracting  nginx-1.6.2\docs\LICENSE
Extracting  nginx-1.6.2\docs\OpenSSL.LICENSE
Extracting  nginx-1.6.2\docs\PCRE.LICENCE
Extracting  nginx-1.6.2\docs\README
Extracting  nginx-1.6.2\docs\zlib.LICENSE
Extracting  nginx-1.6.2\html
Extracting  nginx-1.6.2\html\50x.html
Extracting  nginx-1.6.2\html\index.html
Extracting  nginx-1.6.2\logs
Extracting  nginx-1.6.2\nginx.exe
Extracting  nginx-1.6.2\temp
Everything is Ok
Folders: 12
Files: 27
Size:       3378862
Compressed: 1254010
C:\ProgramData\chocolatey\lib\nginx\tools
ShimGen has successfully created a shim for nginx.exe
The install of nginx was successful.

Chocolatey installed 1/1 package(s). 0 package(s) failed.
 See the log for details (C:\ProgramData\chocolatey\logs\chocolatey.log).

C:\WINDOWS\system32>
```

图 2-8　Windows Nginx 安装完成示意图

接着，我们来创建一个 API——即 JSON 文件，文件的内容如下所示：

```
{
  "led": true
}
```

然后，我们需要去配置 Nginx，使得我们可以直接在浏览器上访问这个 API（JSON 文件）。Nginx 在安装的时候会创建一个默认的配置文件。在不同的操作系统下，默认的 Nginx 配置文件 nginx.conf 所在的目录不同，如在 Windows 系统下是 C:\nginx\conf\nginx.conf，正常情况下，我们执行 nginx -t 就可以获取配置文件所在的目录，结果如下所示：

```
phodal:~ root# nginx -t
nginx: the configuration file /usr/local/conf/nginx.conf syntax is ok
nginx: configuration file /usr/local/conf/nginx.conf test is successful
```

在这里，我们的配置文件是**/usr/local/conf/nginx.conf**。打开相应的配置文件，会

看到默认的配置。为了避免无关紧要的内容出现，移除了默认配置中的注释，默认配置如下所示：

```
worker_processes  1;

events {
    worker_connections  1024;
}


http {
    include       mime.types;
    default_type  application/octet-stream;

    sendfile        on;
    keepalive_timeout  65;

    server {
        listen       80;
        server_name  localhost;

        location / {
            root   html;
            index  index.html index.htm;
        }
    }
}
```

上面的配置看上去稍显复杂，在这里我们可以用一个简化的配置：

```
events
{
  worker_connections 51200;
}

http {
  server {
    root  /Users/fdhuang/designiot/chapter2/www;
  }
}
```

开头的第一行是 events，events 模块中包含 Nginx 中所有处理连接的设置。其中的配置 worker_connections 指的是一个 worker 进程同时打开的最大连接数。在默认配置中的 worker_processes 指定了 Nginx 对外提供 Web 服务时提供的 worker 进程数。在随后的 http 模块中 server 字段则包含虚拟主机的配置。在这里我们只指定了请求达到后的文件目录，即当一个请求过来后，我们会将请求与这个目录进行对比，看是否有这个文件，如果没有的话，会返回一个“404 Not Found”页面。于是，我们需要将 api.json 文件放到**/Users/ fdhuang/designiot/chapter2/www** 目录下。

最后，我们需要启动或重启 Nginx，并在浏览器上访问这个 API。

So，我们可以打开浏览器，访问 http://localhost/api.json 或者 http://127.0.0.1/api.json[7]试试。应该会有如图 2-8 所示的结果。

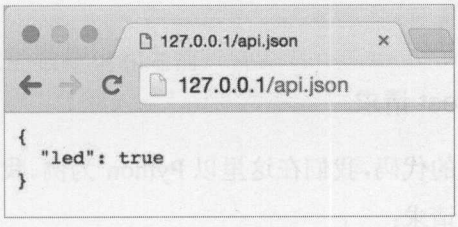


图 2-9　本地环境下的 API 返回结果

打开 access.log 文件，你会看到如下的一条访问日志。

```
127.0.0.1 - - [15/Oct/2015:13:31:51 +0800] "GET /api.json HTTP/1.1" 200
17 "-" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_5) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/47.0.2502.0 Safari/537.36"
```

上面写明了，访问机器的 IP，即 **127.0.0.1**。紧随其后的是我们访问的时间、请求的 URL、协议，这里的 200 指的是状态码，最后的“Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_5) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/47.0.2502.0 Safari/537.36”指的是 User-Agent，上面有操作系统及浏览器相关的信息。

经过这样一个过程，我想你对其中的工作原理已经有所了解了。接着，我们可

7 127.0.0.1 是回送地址，指本地机，一般用来测试使用。回送地址（127.x.x.x）是本机回送地址（Loopback Address）。

以看看通常开发人员是如何获取这些数据的。

2.2.2 获取数据与状态

这时，我们就需要知道如何在我们的客户端（如单片机、手机 APP 等）上发出一个 HTTP 请求。还需要了解如何去从中解析 LED 的状态，即 true。这实际上就是两个不同的过程：

（1）发出请求

（2）解析数据

而根据我们选的不同语言，我们需要依赖于不同的库。在发出 HTTP Request 请求的时候，实现上是用 Socket 来实现这样的请求。接着，我们先看看怎样发出这样的请求。

发出 HTTP Request 请求

为了简化请求过程的代码，我们在这里以 Python[8]为例。我们可以用下面的 Python 代码来实现一个 HTTP 请求：

```
import socket, sys

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("localhost", 80))
s.send("GET /api.json HTTP/1.0\r\n\r\n")

while 1:
    buf = s.recv(1000)
    if not buf:
        break
    sys.stdout.write(buf)

s.close()
```

首先，我们需要先建立一个 Socket 连接，即连接到 localhost:80。然后，发出一

8 Python 是一种解释型、面向对象、动态数据类型的高级程序设计语言。

个请求，请求的路径是 api.json。接着，我们开始接收数据，直到完成。最后，会有同 cURL 一样的结果：

```
HTTP/1.1 200 OK
Server: nginx/1.7.4
Date: Thu, 15 Oct 2015 08:32:54 GMT
Content-Type: text/plain
Content-Length: 17
Last-Modified: Wed, 14 Oct 2015 14:07:43 GMT
Connection: close
ETag: "561e61af-11"
Accept-Ranges: bytes

{
  "led": true
}
```

最后，我们所获取的数据也和之前一样：

```
{
  "led": true
}
```

这样，我们就简单地完成了所需要的一个简单的 HTTP 文件服务器。

2.3 设备状态改变

在我们完成搭建 Web 服务器后，就可以开发我们的硬件端了。

通常，在设计嵌入式系统的时候，我们需要去考虑不同维度的系统要求。由于我们需要网络功能支持，需要有强劲的处理器，以及一个好的协议支持库。除此之外，当需要多任务支持的时候，我们会开始考虑使用嵌入式系统，如 uCOS、vxWorks 等。如果需要考虑更好的扩展性及软件支持，我们就会考虑使用嵌入式 Linux 系统。市面上有不同的基于 ARM 处理器的 Linux 系统开发板，由于种类繁多、系统不兼容等问题，在这里，我们使用树莓派来作为 Demo，演示如何用上面做的 API 来控制一个 LED，以实现一个系统的物联网。

树莓派（英语：Raspberry Pi），是一款基于 Linux[^linux]系统的只有信用卡大小的单板机电脑。它由英国的树莓派基金会开发，目的是以低价硬件及自由软件刺激在学校的基本计算机科学教育。由于 Raspberry Pi 是基于 Linux 系统而开发的，这意味着我们可以在 Raspberry Pi 上运行绝大多数的 Linux 系统上的软件（由于 Raspberry Pi 的处理器是基于 ARM 架构的，会导致一些基于 X86 体系的软件无法在上面运行）。

我们在这里用的是操作系统 Raspbian（基于 Debian 的 ARM hard-float (armhf) 架构）。这意味着，可以用 Python、Ruby、Java、Node.js 等脚本语言来开发我们的 Demo。在完成需要的功能后，我们可以用 Go、C 等编译语言来完善产品的代码。

2.3.1 用 Raspberry Pi 来读取数据

1. SSH 远程登录

在 Raspberry Pi 上运行 Python 脚本和在普通的电脑上运行 Python 脚本没有太大的区别。但是，我们需要先登录到 Raspberry Pi 上。需要将我们的 Raspberry Pi 放在同一网段（如同一个路由器）下。

要登录一台远程机器，通常有 Telnet 和 SSH 两种方式。Telnet 是使用明文传输，即在登录过程中会暴露系统账号和密码等重要信息，因而通常我们会用 SSH 来登录远程机器。SSH 为一项创建在应用层和传输层基础上的安全协议，为计算机上的 Shell 层）提供安全的传输和使用环境。SSH 最初是 UNIX 系统上的一个程序，现在基本上大多数主流平台已经支持这个协议。

在类 UNIX 系统，如 Linux（包括 Ubuntu、Mint、OpenSUSE）系统、Mac OS 系统中，都已经包含了这个软件。Windows 系统用户可以使用 SSH 客户端，如 PuTTY、WinSCP、SecureCRT。如图 2-10 所示是 Windows 操作系统下 PuPPY 软件的截图。

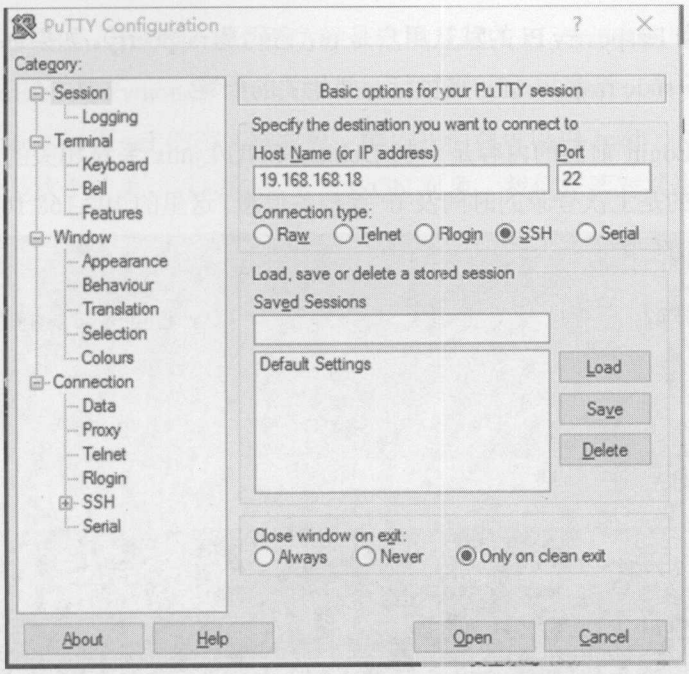


图 2-10　Windows PuTTy 界面图

如果你有 ssh 命令，可以通过下面的命令来登录：

```
ssh pi@192.168.168.18
```

这里的 192.168.168.18 是 Raspberry Pi 的 IP（可以接上显示器并运行 ifconfig 查看 IP 信息，或者通过路由器界面查看 IP）。成功登录 Raspberry Pi 后返回如图 2-11 所示的信息（对于 Windows 系统也是一样的）。

```
fdhuang @test ~ ssh pi@192.168.168.18
pi@192.168.168.18's password:

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Thu Sep 24 15:35:57 2015 from 192.168.168.84
pi@raspberrypi ~ $
```

图 2-11　Windows 下用 SSH 访问 Raspberry Pi

温馨提示: Raspberry Pi 的默认用户是 pi，密码是 raspberry。如果你是首次登录，可以执行一下 sudo raspi-config 来执行一些相关的配置。

在 Last Login 前面的内容是一些 Debian GNU/Linux 系统相关的协议，而 Last Login 则表明的是上次登录的时间及 IP 等相关信息。这里的 192.168.168.84 便是我所使用的计算机的 IP。

如果一切顺利，并且成功启动了 Nginx，就可以在上面访问我们的 Web 服务器了，执行：

```
curl http://192.168.168.84/api.json
```

便会有如图 2-12 所示的结果。

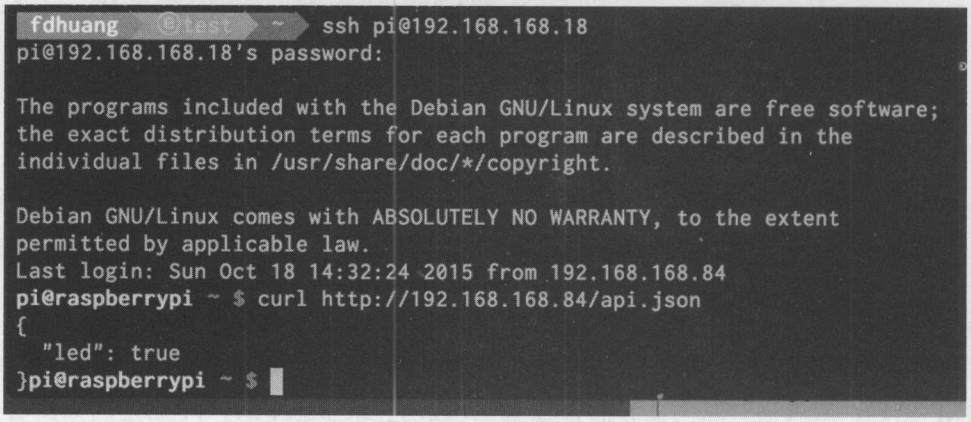


图 2-12　用 curl 脚本获取 API 结果

看样子，可以直接开始写 Python 脚本来实现我们想要的功能了。

2. Python 请求 API

接着，我们在命令行中输入 python 来进入交互环境。我们可以通过执行下面的两行代码来获取 API 的状态：

```
import urllib2
urllib2.urlopen('http://192.168.168.84/api.json').read()
```

执行完后，会返回如图 2-13 所示的结果。

import 是 Python 用来导入模块的方法，这和我们在 C 语言中使用#include 的作用是类似的。urllib2 是 Python 的一个获取 URLs 的组件，它提供了一个 urlopen 函数，用于创建一个表示远程 url 的类文件对象，随后我们便可以用 read()方法读取出 API 中的结果。但是这个结果，返回的是一个 JSON 对象，我们需要对其进行解析。

```
pi@raspberrypi ~ $ curl http://192.168.168.84/api.json
{
  "led": true
}pi@raspberrypi ~ $ python
Python 2.7.9 (default, Mar  8 2015, 00:52:26)
[GCC 4.9.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import urllib2
>>> urllib2.urlopen('http://192.168.168.84/api.json').read()
'{\n  "led": true\n}'
>>>
```

图 2-13　用 Python 来获取 API 的结果

这时，我们需要用到 Python 的 JSON 模块。我们先将第一步的结果赋给一个变量，再对这个变量进行序列化[9]。

```
import urllib2
results = urllib2.urlopen('http://192.168.168.84/api.json').read()
```

接着，我们可以导入 JSON 模块，使用 loads 来返回原来的对象，并返回 LED 对应的值，即 True。

```
import json
json.loads(results)['led']
```

合并上面的四行代码，就会有：

```
import urllib2,json
results = urllib2.urlopen('http://192.168.168.84/api.json').read()
json.loads(results)['led']
```

结果如图 2-14 所示，返回 True。

9 将对象的状态信息转换为可以存储或传输的形式的过程。

```
pi@raspberrypi ~ $ python
Python 2.7.9 (default, Mar  8 2015, 00:52:26)
[GCC 4.9.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import urlib2,json
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named urlib2
>>> import urllib2,json
>>> results = urllib2.urlopen('http://192.168.168.84/api.json').read()
>>> json.loads(results)['led']
True
>>>
```

图 2-14　Python 解析 API 数据

接着，我们就可以进入激动人心的下一步，去控制 LED。

2.3.2　使用 Raspberry Pi 控制 LED

1. Raspberry Pi GPIO 引脚控制

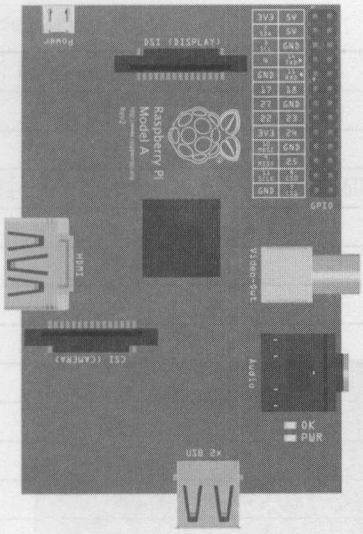
记得在我最开始学 51 单片机、AVR、STM32 的时候都是以点亮 LED 开始，在我学习使用 Raspberry Pi GPIO 的时候，也是以点亮 LED 作为起点。软件领域，人们用的是“Hello,World”，或者 It Works。硬件领域，这个“Hello,World”就是点亮 LED。

图 2-15 是 Raspberry Pi 1 代 A 型的 GPIO 引脚图，图 2-16 是 Raspberry Pi 2 代的 GPIO 引脚图。

考虑到我们需要兼容两个不同的版本，因此我们使用的引脚就限定为 1~26 号引脚。首先，我们需要导入 GPIO 库：

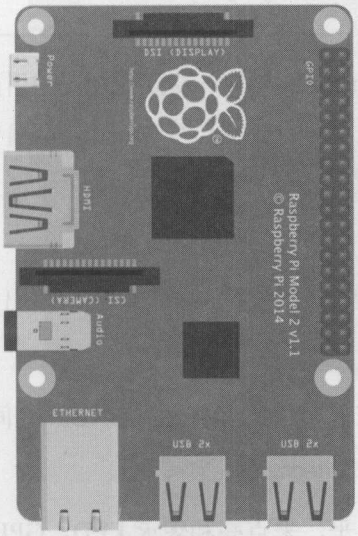
```
import RPi.GPIO as GPIO
```

这里的意思就是为 RPi.GPIO 取一个别名叫 GPIO。由于 GPIO 库有两个不同的引脚编号方式，一种是开发板上的自然编号，另一种是对应于开发板上的 Broadcom 处理器的编号（又称为 BCM GPIO 编号）。通常，我们会使用自然编号，因为我们不需要去记住它们之间的对应关系，并且 Broadcom 的编号方式比较混乱。如表 2-1 所示是三者之间的对应关系。



Pin No.			
3.3V	1	2	5V
GPIO0	3	4	5V
GPIO1	5	6	GND
GPIO4	7	8	GPIO14
GND	9	10	GPIO15
GPIO17	11	12	GPIO18
GPIO21	13	14	GND
GPIO22	15	16	GPIO23
3.3V	17	18	GPIO24
GPIO10	19	20	GND
GPIO9	21	22	GPIO25
GPIO11	23	24	GPIO8
GND	25	26	GPIO7

图 2-15　Raspberry Pi 1 代的 GPIO 引脚图



Pin No.			
3.3V	1	2	5V
GPIO2	3	4	5V
GPIO3	5	6	GND
GPIO4	7	8	GPIO14
GND	9	10	GPIO15
GPIO17	11	12	GPIO18
GPIO27	13	14	GND
GPIO22	15	16	GPIO23
3.3V	17	18	GPIO24
GPIO10	19	20	GND
GPIO9	21	22	GPIO25
GPIO11	23	24	GPIO8
GND	25	26	GPIO7
DNC	27	28	DNC
GPIO5	29	30	GND
GPIO6	31	32	GPIO12
GPIO13	33	34	GND
GPIO19	35	36	GPIO16
GPIO26	37	38	GPIO20
GND	39	40	GPIO21

图 2-16　Raspberry Pi 2 代的 GPIO 引脚图

表 2-1　三者之间的对应关系

引脚号	开发板编号	BCM GPIO
GPIO 0	11	17
GPIO 1	12	18
GPIO 2	13	21

续表

引脚号	开发板编号	BCM GPIO
GPIO 3	15	22
GPIO 4	16	23
GPIO 5	18	24
GPIO 6	22	25
GPIO 7	7	4
SDA	3	0
SCL	5	1
CE0	24	8
CE1	26	7
MOSI	19	10
MISO	21	9
SCLK	23	11
TXD	8	14
RXD	10	15

首先，我们需要使用 GPIO 库的 setmode 方法来设置开发板的引脚模式。当前有两种模式：GPIO.BOARD 为自然编号，GPIO.BCM 为 Broadcom 编号。我们将其设置为 GPIO.BOARD：

```
GPIO.setmode(GPIO.BOARD)
```

接着，我们就可以使用 setup 方法将 5 号引脚的电平设置为输出：

```
GPIO.setup(5, GPIO.OUT)
```

然后，我们需要为 Raspberry Pi 在 5 号引脚和接地（GND）之间接上一个 LED 与其上拉电阻，如图 2-17 所示。

最后，我们就可以设置其为高低电平，来点亮和熄灭 LED。GPIO.HIGH 即为高电平，GPIO.LOW 即为低电平。

```
GPIO.output(5, GPIO.HIGH)
GPIO.output(5, GPIO.LOW)
```

总的代码如下：

```
import RPi.GPIO as GPIO
```

```
GPIO.setmode(GPIO.BOARD)
GPIO.setup(5, GPIO.OUT)
GPIO.output(5, GPIO.HIGH)
GPIO.output(5, GPIO.LOW)
```

或许你再次注意到了：用 Python 写的代码，看上去比较简单明了，可以让我们更专注于想法。

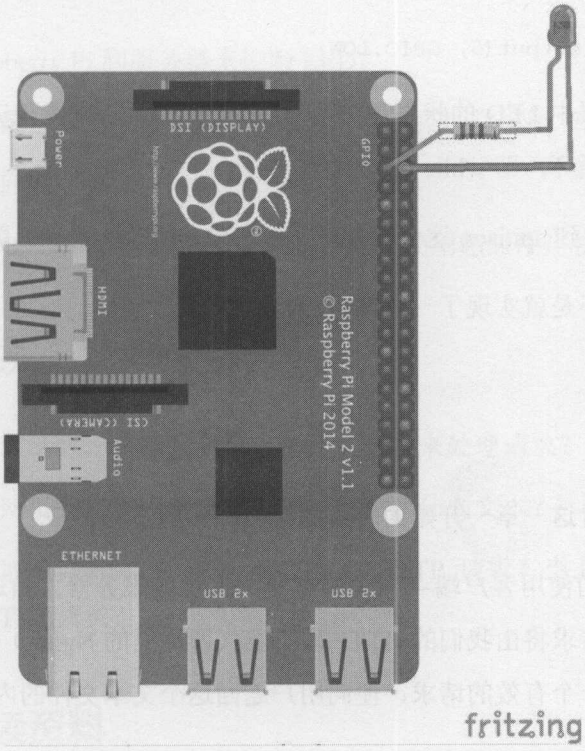


图 2-17　Raspberry Pi 2 连接 LED

2. 使用 Raspberry Pi 控制 LED

现在，是时候让它们可以工作了。我们只需要判断一下 LED 的状态值就可以来开关灯了。

```
import urllib2,json
import RPi.GPIO as GPIO
```

```
GPIO.setmode(GPIO.BOARD)
GPIO.setup(5, GPIO.OUT)

while 1:
    results = urllib2.urlopen('http://192.168.168.84/api.json').read()
    status = json.loads(results)['led']
    if status == True:
        GPIO.output(5, GPIO.HIGH)
    else:
        GPIO.output(5, GPIO.LOW)
```

当返回的结果中 LED 的状态是 True 就点亮 LED，否则就熄灭 LED。我们在最外层加了一个死循环，即 while 1。

现在，让我们到 api.json 这个文件中，去把 true 修改为 false，就可以熄灭 LED。

看，我们是不是就实现了一个简单的物联网！

2.4 小结

再让我们看看这一章一开始的时候的架构图（图 2-1）。

作为用户我们使用客户端——即浏览器，来访问服务器。当这个访问请求来到服务器的时候，请求将由我们的 HTTP 服务器（即这里的 Nginx）来处理。Nginx 识别了这个请求是一个有效的请求，便向用户返回这个文本文件的内容。

在这一章里，我们试图去了解一个基本的物联网是怎样的。我们花了很大的精力去解释 Web 服务器是什么？并用这个 Web 服务器搭建一个简单的文件服务器，来提供一个 API。通过修改这个 API 我们可以控制 Raspberry Pi 上的 LED。而这些内容是后面章节的基础，如果有什么不清楚的，建议查阅相关资料。

在下一章中，我们来分解一个物联网系统，更好地了解系统的层级结构。

2.5 练习建议

如果你使用的是 Windows 操作系统，可以尝试：

（1）按上面的步骤，在 Windows 操作系统上安装 Nginx，来搭建静态文件服务器。

（2）使用虚拟机安装 Ubuntu 操作系统，在上面安装 Nginx，搭建 Linux 服务器上的静态服务器。

（3）连接 Raspberry Pi 到服务器来控制 LED。

如果你使用的是 GNU/Linux 或者 Mac OS，可以直接尝试第二步和第三步。

可能会遇到的问题：

（1）无法使用 80 端口。原因是用户权限不够，使用超级用户权限即可解决。

2.6 问题回顾

（1）访问一个网站时，将由服务器上的什么软件来处理请求？

（2）访问一个网页时，在网页上显示的时候需要什么文本？

（3）用硬件获取数据时，用的是什么类型的 HTTP 请求？当上传数据时，用的又是哪种类型的 HTTP 请求？

2.7 相关阅读资料

[1] [日]上野宣．图解 HTTP [M]．于均良译．北京：人民邮电出版社，2014.

[2] [美]David Gourley，Brian Totty．HTTP 权威指南[M]．陈涓，赵振平译．北京：人民邮电出版社，2012.

[3] 鸟哥．鸟哥的 linux 私房菜[M]．北京：人民邮电出版社，2007.

[4] [英]Simon Monk．Raspberry Pi 开发实战[M]．黄鑫译．北京：机械工业出版社，2015.

第3章

分解物联网系统

本章内容

- 了解物联网系统各个层级
- 了解物联网系统各个层级间的关系及依赖
- 了解物联网系统的组成结构

在上一章中，我们交付了第一个版本的物联网系统。这是一个可以工作的软件，但是通过它我们只能控制单个设备。紧接着越来越多的潜在客户便开始来了解我们的系统，并计划与我们合作，然而当前我们的系统只能控制单一设备。这就意味着我们需要在同一时间支持更多的设备和不同类型的传感器，同时我们也要区分不同的用户的数据。尽管看起来杂乱无章，但我们需要分析并理解其功能，这对于重新设计这个系统将大有裨益。

同样，在开始之前先让我们来看看本章的部分架构图，如图 3-1 所示。

这张图展示了客户端控制硬件设备的一个过程。当我们设计一个物联网系统的时候，就需要考虑用何种设备来控制它们。我们可能在手机上通过 MQTT 协议来连接服务器；在硬件端使用 CoAP 协议来连接服务器；在浏览器端使用 HTTP 协议来连接服务器。这时，我们就需要一个适配层来处理这些不同的协议，只是这个适配层是属于服务端的应用程序的一部分。接着，依据不同的协议我们会有不同的获取数据的策略——如在协调层不断地获取服务器的数据，再将这个数据转发给硬件。

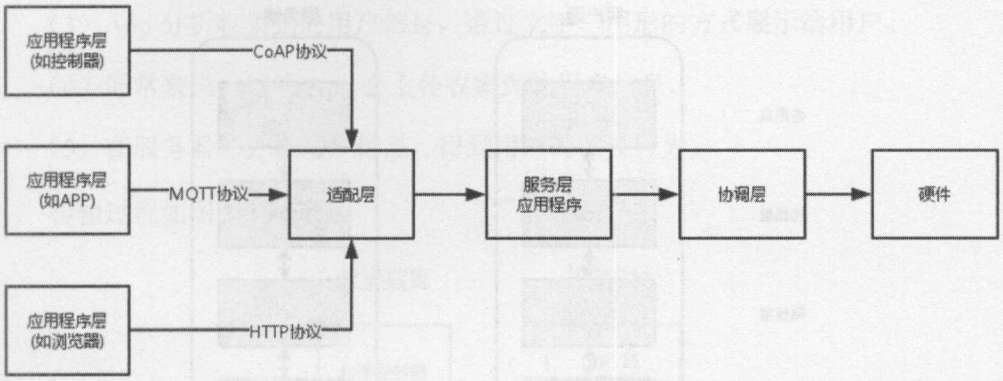


图 3-1　第 3 章理论架构图

3.1　物联网的层级结构

对于设计一个系统来说最简单、有效的（性价比最高）框架模式就是分层，最常见的分层架构模型便是 TCP/IP 协议。在我们的系统里既要考虑低层的硬件实现——传感器、控制器、执行器，也要考虑图形用户界面（在这里我们的图形用户界面主要是 Web 界面）等。传感器的数据会从底层往顶层传输，而执行指令及事件则会由顶层向下执行[1]。

接着我们先来看一下常见的一些应用场景，并识别这些场景下的层级结构。

3.1.1　一个常见场景下的层级结构

如图 3-2 所示便是 TCP/IP 协议的层级表示。

1　《面向模式的软件架构（卷 1）》

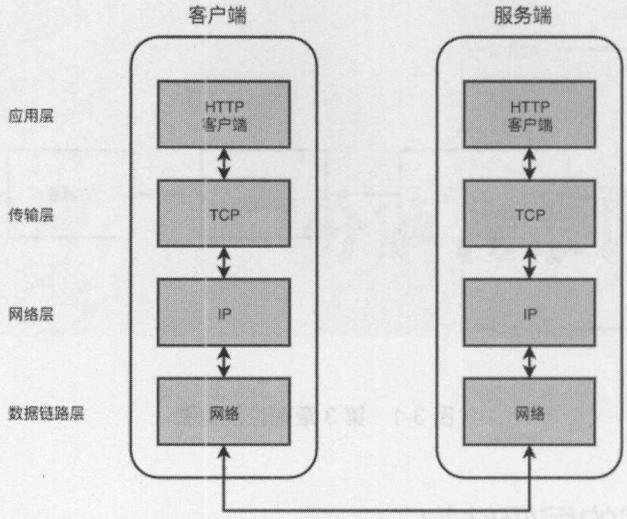


图 3-2　TCP/IP 协议层

如图 3-2 所示，在我们向服务器发送请求的时候，过程是应用层→传输层→网络层→链路层。浏览器向服务器请求的过程，便是由应用层一直往数据链路层请求。当到达链路层时，将发送到右边的协议栈，由下往上传输数据，最后到达服务器的 HTTP 服务端。

当服务器返回响应时，由服务器一层层往下沿相反路径传输，最后达到左上方的 HTTP 客户端。

在我们处理物联网系统的时候，默认就不需要分得这么细致，可以将这部分数据传输的过程变成一个层级。

接着让我们来看看一个手环[2]是如何传递数据的，需要注意的是，在当前的情况下都是单向的，而我们需要的是一个双向传递数据的系统。

（1）通过 GPS、加速度计、心率计、光传感器等来收集用户及环境数据，并存储到 Flash（存储器）中。

（2）当手机连接上手环时，App 通过蓝牙通信读取 Flash 中的数据，存储到手机中。

2 这里并没有使用智能手环，因为当前市面上的手环并不智能。

（3）App 分析收集到的用户信息，通过文字、图形的方式展示给用户。

（4）通常来说，这些 App 会上传收集到的用户信息。

（5）在服务器端分析用户信息，得到用户习惯及行为。

传输过程如图 3-3 所示。

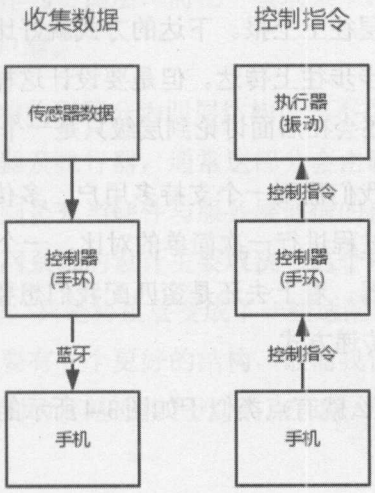


图 3-3　数据传输图

我们可以先对此进行简单的分层：

（1）硬件层。在这里指的便是手环，手环包含了传感器、控制器、执行器。

（2）协调层。在只考虑手机传输数据到服务器的这一步时，手机 App 在这里充当的是协调作用。

（3）用户应用层。在只考虑手环展示用户数据的情况下，App 在这里就是应用层的实现。

（4）制造商应用层。对于制造商来说，记录用户、分析数据的服务器等就构成了其应用层。

这样的应用实例也接近于我们需要的物联网层级结构。只是在这里并没有考虑到一个协调层可能有多个硬件，如传统蓝牙可以支持多达 7 个设备，在这时协调层

就显得很重要。

我们这里的层级划分主要根据数据流，而上一章则是以控制为核心的层级结构。

3.1.2 理想的物联网层级结构

通常来说，在物联网应用中主要就是控制和数据流两种方式。指令需要一层一层往下下达，或者一层一层往上上报。下达的方式就好比是中央到地方，再到具体的地方，而后再将结果一步步往上传达。但是要设计这样的层级架构在一开始不是一件简单的事，而且我们还会在后面讨论到层级只是一个理想的参考模型。

让我们再看看需求，我们需要一个支持多用户、多传感器的物联网系统。我们可以将之与上面的手环的过程进行一次简单的对比，一个用户就相当于一个 App，而一个手环上有多个传感器。看上去还是蛮匹配我们想要的功能的，除此之外，我们还需要考虑控制指令的传递方式。

如果不考虑硬件层，那么就有点类似于如图 3-4 所示的传统的 C/S 结构（事实上，加上硬件层也有点像）。

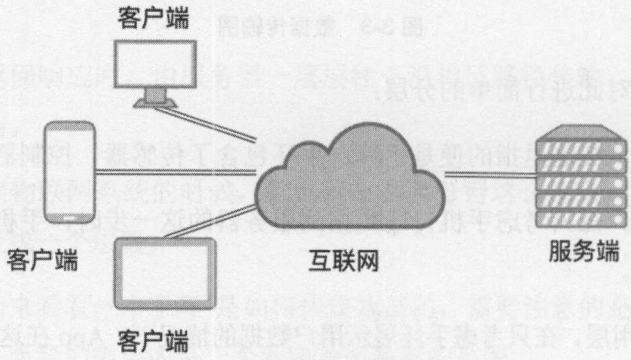


图 3-4 C/S 结构

但是，与传统 C/S 结构不同的是：

（1）传统的服务端一直是处于被动模式，即当服务端接收到一个请求时，才做出相应的反应。但是对于物联网系统来说，服务端需要主动推送消息。

（2）传统的结构更像是以服务端为中心的结构，而物联网的结构则需要侧重于以客户端为中心，服务端更多是辅助作用。

（3）传统的客户端在与服务端失去连接的时候便不可用，而对于物联网来说这种情况是不允许的，客户端与服务端失去连接的时候也需要在某种程度上可用。

在上面的手环中手机作为协调层，而在一些场景下，硬件层包含了网络连接的功能，直接变成了一个客户端。

如图 3-5 所示，我们将物联网分为四层结构，而不是三层结构[3]。硬件层包含了数据众多的传感器、控制器及执行器，通常这部分会由硬件人员与硬件开发人员一起协作和开发。而协调层则是充当硬件与服务层通信的桥梁，这是在系统中需要**特别考虑**的部分，一个物联网系统的设计主要**取决于这个层级**。而服务层的核心是传统的 Web 应用程序的结构，只是协议层变成了一些适配器，我们需要支持不同的协议，这导致了在这个层需要有一个更好的结构，故而我们建议使用**六边形架构**。而在实际中，用户最后接触到的便是应用程序层，在这一层中需要有很好的用户体验设计及流畅度。

我们可以将之总结为表 3-1。

表 3-1 理想的物联网层级结构

层级	作用	与下一层级的连接方式
硬件层	获取、发送传感器数据，执行指令	无线和有线两大类
协调层	协调硬件层与服务器的通信，并负责处理部分数据	网络连接及硬件层的连接方式
服务层	以视为服务器层	网络连接
应用程序层	为用户提供交互功能	网络连接

3 感知层、网络层、应用层的三层结构只是一个理想的模型，并没有充分考虑在实际实践中可能遇到的问题。

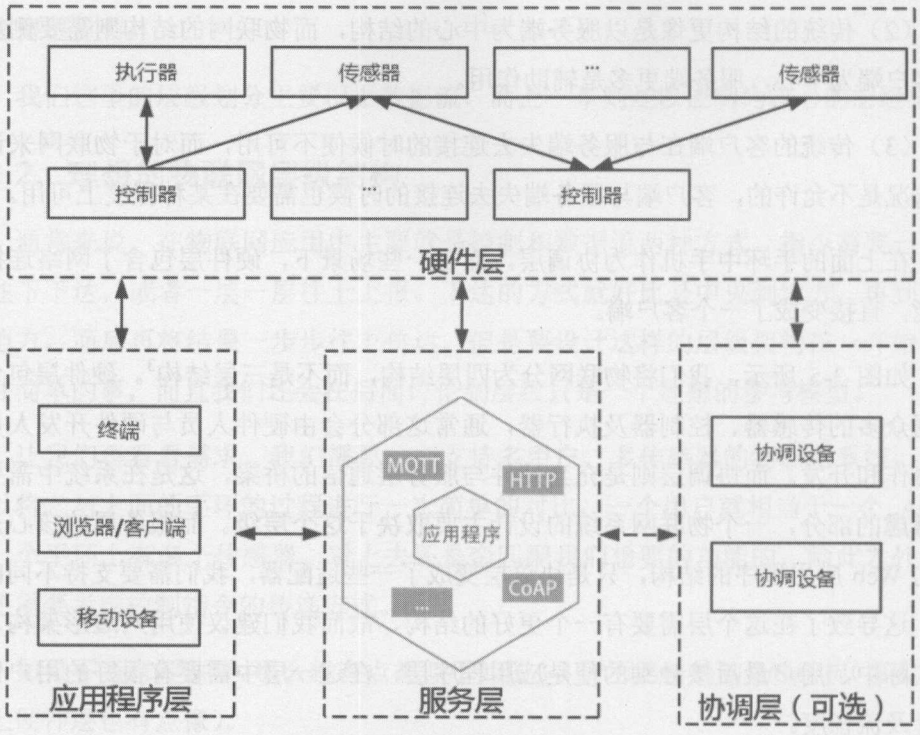


图 3-5　理想的物联网层级结构

在我们设计这样的系统的时候要考虑是否更细分层级，或者合并层级。如手机的 App 充当了协调层与应用程序层，而在我们后面讨论的一些例子中，如 Raspberry Pi 之类的卡片电脑则可以包含三个或者四个层级。 因而，我们选择的硬件设备也决定了整个系统的层次关系。

（1）如果选择了 Arduino、51 单片机，那么我们可能还需要一个协调层来负责调度功能，这个协调层会在这里处理部分数据。

（2）如果选择了 NodeMCU、Spark 这种带网络功能但是不运行操作系统的开发板，那么我们可能会把收集数据部分的功能放到服务器上。

（3）如果选择了 Raspberry Pi、WRTNode 这种类型带操作系统的开发板，那么我们可能会在开发板上处理数据、收集数据，存储数据。这样的系统包含了四个层级，当然这样的系统在个人开发上比较容易，而在协作上则会出现小问题。

除此之外，我们还需要考虑如何连接各个层级。

3.1.3 与真实世界交互的物理层

物联网从理论上延伸了机器的世界，从虚拟世界到真实世界。因而它也像这个世界的生物一样，有自己的感知，有自己的响应。传感器便是它的感知层，控制器相当于它的神经中枢，执行器可以相当于它的手脚嘴等。

回顾我们初学嵌入式系统的时候，首先就是用控制器对执行器进行操作。从简单的控制 LED、控制电机、控制 LCD，我们都是在使用不同的执行器。随后，我们开始使用不同的传感器：温度传感器、温湿度传感器等。我们都在以某种方式与真实的世界交互。

1. 物联网的感官——传感器

传感器是物联网的一个重要组成部分，它可以让机器感知真实的世界。在制作一些智能设备的时候，如智能花盆，我们就需要一个土壤传感器来检测土壤的湿度，当湿度不够的时候，控制器就会下达浇水命令。传感器可以说是决定一个物联网是否智能的关键。

人们设计了一个新的传感器，这也意味着这个传感器可以感知真实世界。

我们可以将传感器与人的感官相比拟：如视觉。常见的有图像传感器、光敏传感器，最简单的如光敏电阻，可以用于感知环境光的强弱，复杂的如用于手机摄像头上的图像传感器，可以用于拍照。而在一些人类不可见的光中，如红外线传感器、紫外线传感器，也属于光敏传感器。

但是如果以此分类就意味着，有大量的传感器无法归类到视觉、听觉、嗅觉、味觉、触觉中的某一类。通常我们在使用的时候会以应用来分类传感器，如 LM35 会被分到温度传感器。一种比较简单的方法是根据它们要测量的量对其进行分类，如物理类的力、热、光、电、磁和声等，还有化学类、生物类。在一般的领域里，我们要测量的都是物理量。如图 3-6 所示是一个光敏电阻。

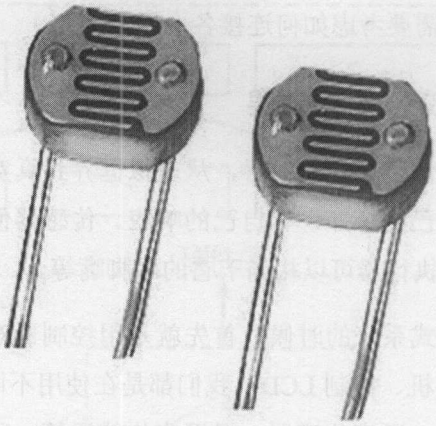


图 3-6　光敏电阻

其原理是：当有光线照射时，电阻内原本处于稳定状态的电子受到激发，成为自由电子。光线越强，产生的自由电子也就越多，电阻就会越小。通过这样的关系，我们就可以获取到光照强度。

在使用一个传感器之前，我们需要先知道需要测量什么样的值，我们想去知道不同状态下这个值会是怎样的。依据不同的环境，如高温、防水，我们又需要不同形状或者不同的传感器。**先去定义我们的问题，随后才能知道我们需要什么。**

2. 物联网的心脏——控制器

或许，你同我一样在最开始学习嵌入式系统设计的时候，是从 51 单片机开始的。这是一个经典的微控制器，各高校都使用其作为一个代表来进行理论学习，然而这个单片机无法应对我们对处理性能的要求。而且有足够的理由相信，你和我一样也不喜欢在上面编程，在上面编程体验一点儿也不好。

在我结识了 Atmel ATmega16 之后，我就发现了一个更好的世界。在上面除了可以使用汇编和 C，还可以使用 C++，或者 Ada[4]语言。随后，我看到了 Arduino[5]，它

4 Ada 是一种程序设计语言。它源于美国军方的一个计划，旨在整合美军系统中运行着的上百种不同的程序设计语言编写的程序。

5 Arduino 是一款便捷灵活、方便上手的开源电子原型平台，包含硬件（各种型号的 Arduino 板）和软件（Arduino IDE）。

创造了一个新的世界。它让我意识到，电子产品也可以像积木一样组建。尽管在那之前，我已经意识到了模块化的重要性。但是，我从来没有想过电子产品还可以这样玩。即使后来开始使用 ARM 处理器，也没有如此的兴奋。让人兴奋的还有 FPGA，这是一个美妙的产品，我们可以在集成电路中设计电路!! 真想分享所有自己知道的那些控制器，但是控制器实在太多了。

控制器作为我们编程的对象，它是传感器和执行器的协调装置。通常我们会赋予控制器三个基本功能：控制、保护与监测，但是最常见的就是控制。控制器将接收传感器的信号，将相应的控制信号发送给执行器，来改变物体的状态。例如在上文提到的智能花盆的例子中，控制器将接收传感器的数据，即当前花盆的温度。当湿度不足时，将发送信号给执行器，执行器将会浇水。

对于普通的用户、开发者而言，在使用某个控制器的时候，通常会选择某个开发板。开发板是用来进行嵌入式系统开发的电路板，通常会包括中央处理器、存储器、输入设备、输出设备、数据通路/总线和外部资源接口等一系列硬件组件。针对同一个处理器，市面上会有不同类型的开发板。如图 3-7 所示，是目前国内外比较受欢迎的 Arduino 系列开发板。

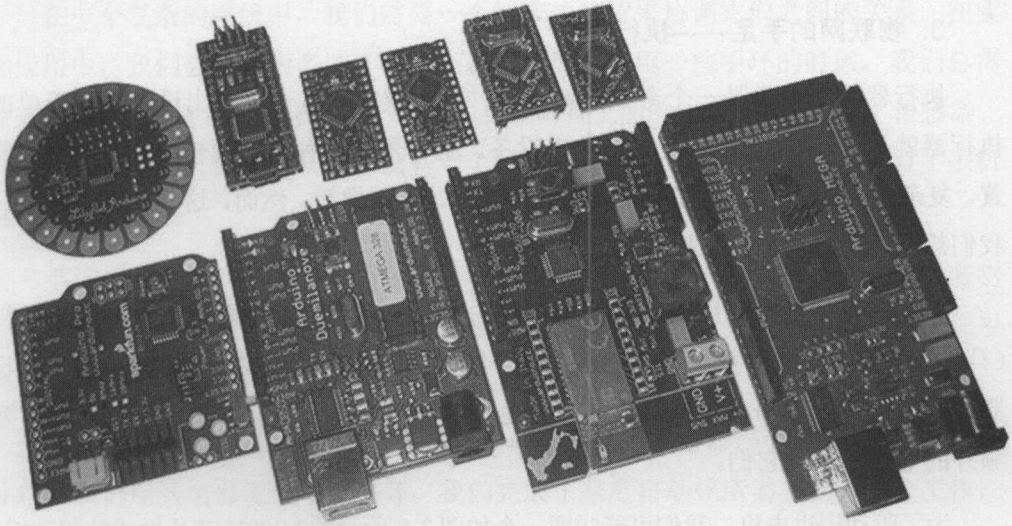


图 3-7　Arduino 微控制器家族

在学习 Arduino 时，我们推荐使用 Arduino UNO 开发板，因为有大量现成的模块。在制作可穿戴设备的时候，我们会考虑使用 Lilypad，图 3-7 左上角的圆形开发板。当我们需要在一个狭小的空间中使用处理器的时候，可以考虑使用 Arduino Pro Mini。尽管我们使用的是 Arduino UNO 来开发原型，但是**不需要任何的代码改动**，就可以在 Arduino Pro Mini 上运行，唯一需要做的便是在烧录的时候，选择该开发板。

在我们选择控制器的时候，同样也需要考虑要处理的数据量的大小？需要在多少时间内处理完？是否需要及时响应？如果我们需要处理大量的数字信号，可能更需要一个 DSP 微处理器，而不是一个普通的微处理器。如果我们需要的是大量的数据处理，可能要考虑将这些数据放在一个集群上处理，而不是放在本地。如果我们不需要及时响应，但是需要机器学习的相关算法，可以用运行着 Linux 系统的 ARM 处理器来解决。

同时，我们还需要考虑的一个问题是：是否有一个好的集成开发环境。好的集成开发环境，如 Keil C51，包含着编辑器、解释器、编译器等功能，它甚至还包括一个操作系统（RTX-51），其他的如 Freescale Kinetis 系列的某些处理器的 IDE，可以直接在线调试。

3. 物联网的手足——执行器

执行器看上去像是一个奇怪的新概念，有时候它就是一个输出设备。最简单的执行器就是 LED，LED 也是一个输出设备。执行器是一种将能源转为机械动能的装置。复杂一点的输出装置有 LCD，可以显示文本、图像等。然而，这些都只是输出。我们想要的更多的是，操作真实世界的物体。

上文中提到的浇水用的水泵也是这样的装置。常见的执行器还有电机、电动机（马达）和步进电机。如果我们想打造一个机器人，那么这些就是不可缺少的。执行器在嵌入式系统中扮演着非常重要的角色。而对于大多数执行器来说，我们都需要额外的电路来驱动它们。

为了驱动步进电机，我们可能需要一个如图 3-8 所示的 L298N 芯片做成的驱动板。

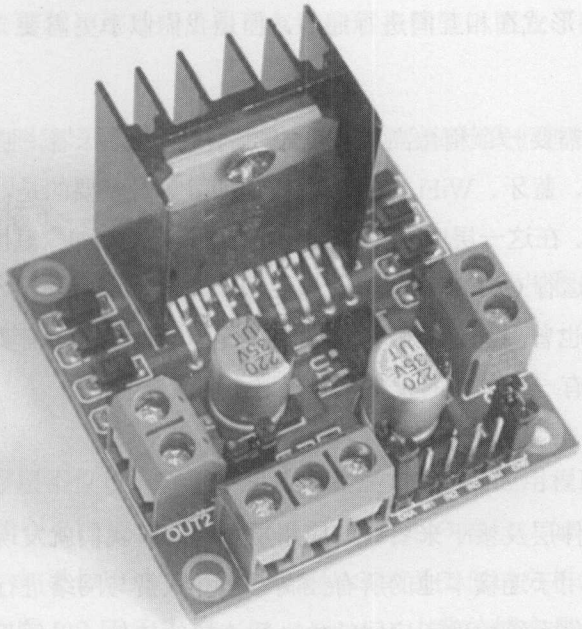


图 3-8　L298N 驱动板

3.1.4　物联网的神经中枢——协调层

在一个复杂的系统中，我们需要一个特殊的装置来协调它们之间的关系。在实际应用中，我们会发现很多这样的例子。当我们在使用一些手环的时候，我们总需要去连接我们的手机来查看数据。手环本身就具备上面的物理层的三要素：传感器、控制器、执行器。但是，我们并没有一个好的渠道来管理它们。因此，就有了手机这个协调层。

手机本身就是一个很好的协调层，手机支持网络功能、蓝牙功能，这意味着它不仅可以和服务器通信，也可以和硬件层通信。而通常，对于协调层的硬件来说也需要这样的功能。

我们已经设计了一个硬件层，我们已经解码了红外遥控器，可以直接控制空调、冰箱，我们用继电器接上了浴霸等。我们完成了家庭自动化所需要的大部分工作，然而由于房间限制，我们在多个房间中使用了多个控制板。尽管我们可以使用蓝牙、

Zigbee、WiFi 等形式在相互间进行通信，但是我们似乎更需要一个核心装置来管理这些。

这时候，就需要一个稍微高端一点的控制板。这意味着，它可能需要支持多种通信方式：红外、蓝牙、WiFi、2.4G 无线通信等。最主要的是，它需要协调这些设备一起完成工作。在这一层中，它会以不同的形式出现，如家庭网关、电脑主机等，过去我一直想用运行 OpenWRT Linux 路由器来完成这个功能，后来发现它的主要功能应该是路由。也曾用过手机来充当这个职责，但是我们还需要拿手机来接电话。所以，我们应该有一个单一的设备来履行这个职责。

一个中间件

与上面的硬件层及接下来要讲的应用层一对比，我们就发现了协调层的核心用途：中间件。它用于连接本地的所有控制器，并负责与网络进行通信。中间件本身的定义就是：提供系统软件和应用软件之间连接的软件，以便于软件各部件之间的沟通。

我们在不同的环境下安装不同的组件，由于会遇到不同的障碍物，我们可能会考虑不到通信协议、设备。这就意味着，假设我们的中间件不具备某些特别硬件，它应该也要具备接口能力。而所谓的接口能力，说的不仅仅是在中间件上实现硬件的功能，在多数情况下我们无法做到这一点。通常来说，这个接口能力指的是，我们可以在这里借助于其他设备来完成。

今天一个新的事物流行可能只需要短短的几个小时，如 iPhone 的出现，意味着虽然从过去到现在我们一直是市场的领导者，但是不意味着永远都是。新的设备可能会带来新的协议，如 Nest 带来了 Weave 协议，这时我们的系统还能正常工作吗？这时候我们需要怎么做？

如果我们想去控制旧空调，但是我们的手机不带有红外功能，我们可以借助于某个带有蓝牙和红外的设备来帮助我们。对于中间件来说，它需要有这样的机制，需要开放一些接口，而不是只满足现有的设计。硬件与软件的很大不同之处在于，软件的更新，对于用户来说可能没有花费，但是即使你免费更新硬件，用户也不一

定有时间让你上门去更新。这带来了极不友好的用户体验，阻止了产品的进一步流行。

协调层通常会依照物理层的设计，而它的设计也会依赖我们想要的通信模型。

3.1.5 物联网的核心——应用层

这里我们说的应用层是由应用程序层与服务层组成的。应用程序层主要集中于用户与后台管理，而服务层则关注于为应用程序层与硬件层提供一个好的接口。

1. 应用程序层

这里的应用程序层不仅仅局限于手机应用程序、PC 端应用程序，还应该有网页，有时我们还需要一个仪表盘（Dashboard）来向用户展示。如手环的手机端便是属于这类的应用，如果你用过 Google Analtyics 或者类似的分析工具，那么这也属于应用程序层。如图 3-9 所示是 Google Analytics 中我的博客最近一个月的“受众群体概览”。

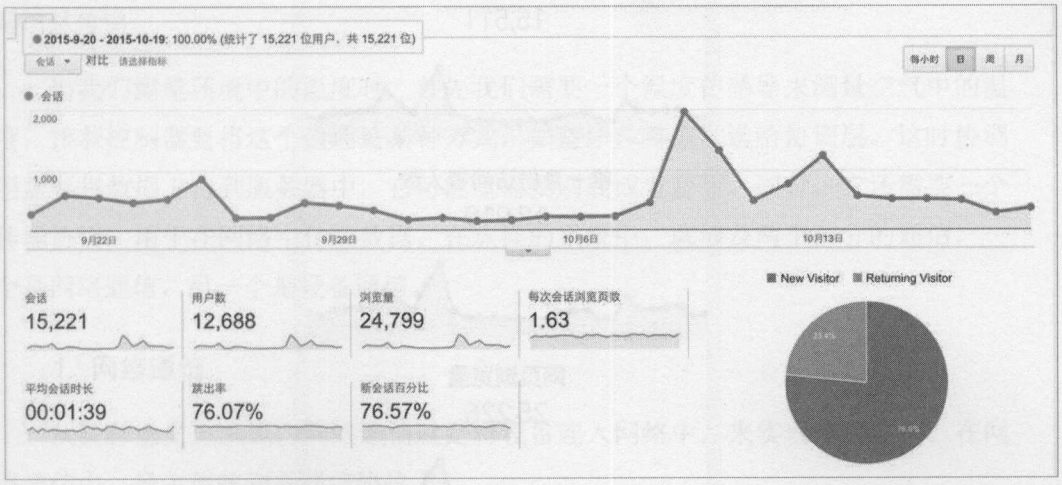


图 3-9 Google Analytics 受众群体概览

通过观察，可以发现 10 月 9 号流量异常，我们就需要对其进行分析，了解其因果。如果放在我们的物联网应用中，它就是我们的仪表盘。将图 3-9 换成电费的使用

情况，我们就可以分析为什么这一天使用了这么多电，是因为我们忘了关空调？还是因为电器故障？

如果这一层级实现机器学习，那么我们就不需要天天观察这些数据。在应用不够智能的时候，我们就需要可视化的数据来帮助我们。

2. 服务层

由于我们不可能只采用单一的协议，例如在上一章中我们使用 HTTP 协议来搭建一个简单的物联网，为了兼容更多的设备，我们需要支持更多的协议。如图 3-10 所示是 Google Analtyics 的移动应用中的“受众群体概览”。

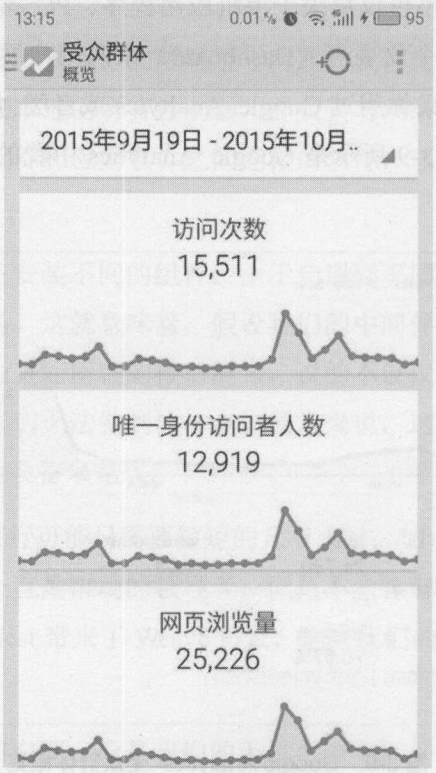


图 3-10　Google Analytics 移动版

针对 Web 客户端，我们提供的 API 使用 HTTP 协议，但是针对移动端，我们可

能会使用 WebSocket 协议。同样，在硬件领域也是如此，如果我们的设备受到本身的资源限制，那么我们可能就不能使用 HTTP 协议，转而考虑 CoAP 协议。

服务层提供的往往是客户端与数据库间的服务，广泛地说便是用户与数据库的服务。我们在服务层所做的就是一些数据的相关处理——创建、读取、更新、删除，即 CRUD。在这个过程中，我们会进行一些相关的数据转换。例如对于一个普通用户，我们应该只返回它相关的数据。如果这个用户想要的是今天的数据，那么我们就要进行一些筛选。数据以某种形态存入数据库，再以某种形态展示给用户。

服务器只是在这中间做一些特定的处理，然而由于我们的业务都集中于这一层，这一层更多的是业务相关知识的一些体现，这也就是为什么这一层复杂的原因，毕竟业务是变化的。

3.1.6 通信

在每个系统中，通信都是一个重要的组成部分。这主要是因为一个系统通常是由多个个体组成的，它们之间自然而然地需要去联系彼此。通信从某种意义上来说，就是信息传递。

当我们测量环境中的温度时，首先我们需要一个温度传感器来测量空气中的温度，接着控制器要将这个值通过某种方式，如蓝牙，将值发送给协调层。这时协调层需要将数据上传到服务器中，它可能就需要网线或者蓝牙。同时，它还需要一个传输协议，用于在网络中传输数据。在这样的系统中，就涉及两个部分的通信，一个是网络通信，另一个是设备通信。

1. 网络通信

从某种意义上来说，网络通信就是将设备连入网络中，来实现信息交换。在网络通信中，最主要的便是通信协议。

在当前的互联网中，最常见的通信协议是 HTTP。在当前的物联网系统中，最常见的通信协议似乎也是 HTTP，原因或许是因为它在互联网协议中比较受欢迎。而且对于现有的控制器来说，已经有各种各样的以太网模块，如 W5100、ENC28J60。从

某种程度上来说，不改变当前的网络模块的硬件，可以说是一种趋势。因此，我们会更多地关注于网络协议。常用的物联网协议有 HTTP、MQTT，CoAP 协议对于资源有限的设备来说是另外一种趋势。现有的硬件也可能会采用 UPnP 或者 XMPP 协议，但是并不推荐。我们可以对 HTTP、MQTT 和 CoAP 做一些简单的对比（当前也有其他的一些协议处于研发期，如 Google 的 Weave 协议），如表 3-2 所示。

表 3-2　网络通信协议对比

协议	CoAP	RESTful HTTP	MQTT
传输	UDP	TCP	TCP
消息模式	请求/响应	请求/响应	请求/响应 发布/订阅

大多数物联网设备连接到计算机的方式是无连接的方式，即只在需要的时候和服务器连接。而 TCP 协议则是面向连接的协议，这就意味着当我们的设备连接到服务器时，需要建立一个新的连接。并且在连接的时候，TCP 协议需要三次握手，这意味着我们会花费更多的网络开销。相比之下，UDP 协议在同样性能的机器上可以处理更多的请求，同样的数据量下需要的数据包更小。

在本书创作期间，CoAP 协议并不支持发布/订阅模式。CoAP 是一种应用层协议，它运行于 UDP 协议之上，它简化了 HTTP 协议。与 HTTP 协议相比，对于同样大小的消息，CoAP 传输的内容更少，对于那些资源有限的设备来说，它是一个不错的协议。MQTT 协议最开始用于消息通信协议，它的主要特性是发布/订阅消息模式。这就意味着，它可以成为一个消息平台。关于这些协议，我们会在后面的章节里进行详细的介绍。

通常来说，我们应该先设计通信模型，依据通信模型选择协议，再依据所需要的协议选择开发板。这是系统的设计过程，而在实现的时候往往会从底层往上层设计。

2. 设备通信

两个或多个嵌入式设备间通信有太多的方法。连接手机和开发板，我们可以使用 WiFi 和蓝牙。在一些特定的手机上可以用 NFC、OTG、红外。或许你也曾经听过

Zigbee、Z-Wave、Thread、6LoWPAN 等短距离无线通信协议/技术。然而，由于当前项目的限制，我并没有机会一一尝试这些通信技术。

从技术实现及普及率上来说，WiFi 和蓝牙都是不错的选择。近年来流行的一些带无线网络模块的开发板中都出现了 ESP8266 无线模块，如 NodeMCU[6]，其性价比比蓝牙高很多。大多数时候，我们设计一个嵌入式系统的时候，价格是不得不考虑的因素，而开发难度也是这样的因素。

在我们的系统中，通常会由高一级的设备来收集低一级设备的信息。这种层级结构，有点类似于省、市、区这样的概念。这就意味着，上一级的设备可以与其拥有的下一级设备通信，它们需要更多的通信方法。

3.2 小结

现在，你是不是已经更了解如何去设计一个物联网系统了？下面让我们来看看与本章开头相反的架构图，如图 3-11 所示。

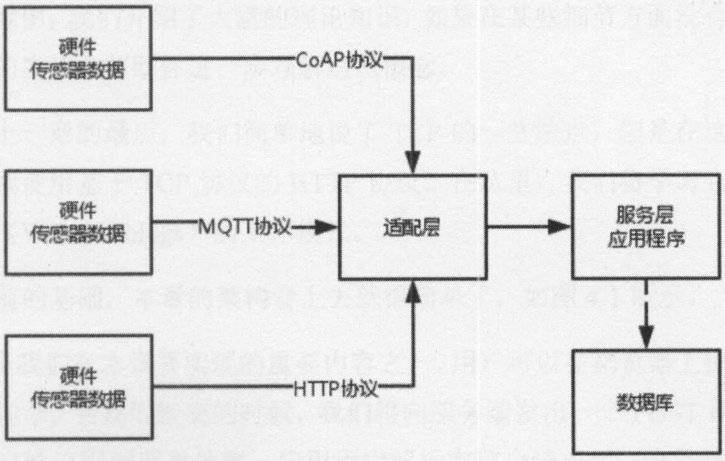


图 3-11 第 3 章理论架构图 2

我们也使用同样的方式来向服务器发送传感器数据——不同的硬件可能使用不

6 NodeMCU 是一个开源的基于 Lua 脚本语言的物联网平台。

同的协议，在服务层同样也有相似的适配层来对它们进行处理。

在这一章中，我们了解了如何去设计一个物联网系统，以及这个物联网系统的层级结构。在分解这些结构的过程中，你或许已经了解了在当前的情形下，你需要怎样的设备，需要如何选择通信协议。

下一章，我们将以此为基础，设计一个更细致的物联网。

3.3 相关阅读资料

[1] [德] Frank Buschmann. 面向模式的软件架构卷一: 模式系统[M]. 袁国忠译. 北京：人民邮电出版社，2013.

[2] [英]Martin Fowler. 企业应用架构模式[M]. 王怀民，周斌译. 北京：机械工业出版社，2004.

[3] [英] Simon Brown. 程序员必读之软件架构[M]. 邓钢译. 北京：人民邮电出版社，2014.

第4章

基于 Web 的物联网系统

本章内容

- Web 应用的基本框架
- RESTful 的基本概念
- 如何开发一个基于 Web 应用的物联网系统

在上一章中，我们介绍了大量的理论知识，如果在某些细节方面没有理解清楚，我相信本章的实战会帮助你进一步理解这些概念。

尽管在上一章的最后，我们简单地说了 TCP 的一些缺点，但是在这一章中，我们还是会继续使用基于 TCP 协议的 HTTP 协议。在这里，我们将学习到平时见到的那些物联网（Web of Things）的工作模式。

有了之前的基础，本章的架构看上去就很简单了，如图 4-1 所示。

图 4-1 是我们在本章要实现的重要内容之一，用户可以在浏览器上提交表单数据，即开关控制指令。当我们提交的时候，我们将向服务端发出一个 POST 请求，接着这个请求将会交给应用程序来处理。应用程序解析完这个请求后，会将这个指令存储到 MongoDB 中。

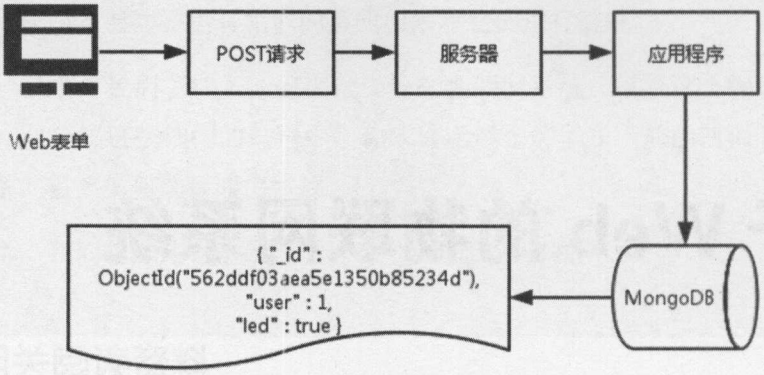


图 4-1　第 4 章程序架构图 1

4.1　Web 应用架构

Internet 是由网络与网络所串连成的庞大网络，链接全世界几十亿个设备。而这些设备，即计算机，都有一些相似之处。如我们经常看到 LNMP（由 Linux + Nginx + Mysql/MariaDB+ Perl/PHP/Python 组成）或者 LAMP（由 Linux + Apache + Mysql/MariaDB+Perl/PHP/Python 组成）这样的字眼，相信看了它的组成，你就明白了它的组成。

对于一个 Web 应用来说，首先它需要有一个操作系统，无论是 Windows、Linux、UNIX 或者是 Mac OS 都是可以的。而操作系统在某种程度上可能会局限系统的扩展性。如对于 Linux 系统来说，要搭建 Linux 服务器集群系统是一件很轻松的事。但是对于某些系统，如 Windows，可能会存在某些困难。

本章所需的软件清单：

- Node.js：一个开放源代码、跨平台的、可用于伺服器端和网络应用的运行环境，由 JavaScript 编写而成。
- MongoDB：一个基于分布式文件存储的数据库。
- Express：一种保持最低程度规模的灵活 Node.js Web 应用程序框架。

4.1.1 MVC

最常见的 Web 应用的架构就是 MVC，它把软件系统分为三个基本部分：模型（Model）、视图（View）和控制器（Controller），如图 4-2 所示。

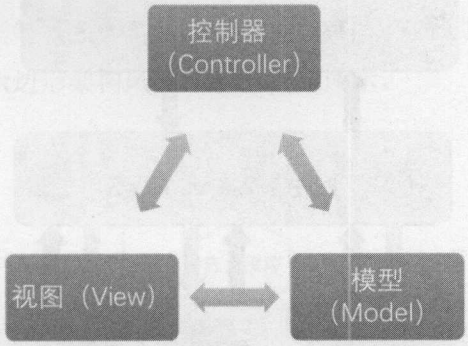


图 4-2 MVC 架构图

这三种组件的定义如下。

（1）控制器（Controller）负责转发、处理请求。用于控制应用程序的流程，并且组织不同层级的关系。如我们访问某网站的“/”和“/blog”的 URL 时，它们由不同的控制器来管理，并且返回不同的结果。

（2）视图（View）即用于显示，只是有**目的地**显示数据。如当某用户访问自己的个人资料页时，只显示自己的个人资料，而不显示其他人的资料。

（3）模型（Model）即进行数据管理。通常，在这个层级会封装与应用程序业务逻辑相关的数据，以及对数据的处理方法。如对于从数据库中获取某个用户的资料，在这一层级会封装对应的处理方法。

而到了应用的时候就不是这么简单了，如图 4-3 所示的架构是对 MVC 架构的一种演变。在这个实现中出现了服务层——介于模型层与视图层之间。

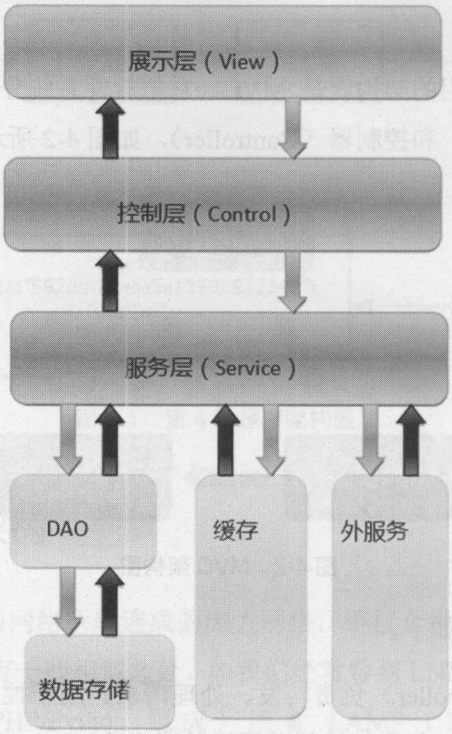


图 4-3　真实应用下的 MVC 架构

这时不得不提及《企业应用架构模式》，Martin Fowler 在书中提到了 Web 应用的三层结构，即表现层、领域层与数据源层。该书中文译版出版的时候是 2004 年，对于不同层级的职责分配如表 4-1 所示。

表 4-1　Web 应用的三层结构

层次	职责
表现层	提供服务、显示信息、用户请求、HTTP 请求和命令行调用
领域层	逻辑处理，系统中真正的核心
数据源层	与数据库、消息系统、事物管理器和其他软件包通信

4.1.2　领域与适配器层

对应于我们的情况，则可以分成捎带耦合的三层结构：领域与适配器层、数据持久化层、视图与应用层。你可以将之对比为 MVC 结构的模型（Model）、视图（View）

和控制器（Controller）。

只是这个控制器需要更多的适配器，因此我并不将它视为一个单纯的控制器。

领域与适配层，可以看作这里的业务的核心。我们会在这里放置大量的逻辑，如用户授权、记录数据等。在这里我们会以 API 的形式提供服务，并且需要支持多种协议，于是就有了六边形架构风格[1]，如图 4-4 所示。

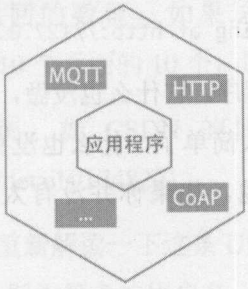


图 4-4　六边形架构风格

实现这里的应用程序时会存在一个领域层，而周围的 MQTT、CoAP、HTTP 等协议与服务则属于适配器。在完全应用程序的功能后，我们只需添加相应的适配器就可以完成这个功能。然而，要一次做到这样的结构显然是不可能的。当系统越来越庞大的时候，我们也可以将一个个适配件以微服务的形式抽取出来。在这里第一步要做的，显然是要先做一个 API 来提供服务，然后再抽取出相应的适配层。

简单来说，这一层就是一个简单的 HTTP 服务。

4.1.3　最小的 HTTP API

相信你已经厌烦了上面的理论，让我们再来写一个“Hello,World”吧！接下来，我们将用 Node.js 与 JavaScript 来写一个“Hello,World”。别担心，它并没有你想象得那么难，参考“附录 B”来了解这门语言。在那之前，如果你还没在你的平台上安装 Node.js，那么你可以参考官方网站：https://nodejs.org/。

1　来源于 Vaughn Vernon 的《实现领域驱动设计》。

下面是一个 Node.js 写的“Hello,World”的示例。

```
var http = require('http');

http.createServer(function (request, response) {
  response.writeHead(200, {'Content-Type': 'text/plain'});
  response.end('{"led": true}');
}).listen(8080);

console.log('Server running at http://127.0.0.1:8080/');
```

这里只是一个简单的应用程序，它什么也没做，只是重写了一个 nginx + api.json 写的那部分代码。这样的服务太简单了，什么也没有做到。但是，我们应该怎样去设计一个 API 就是复杂的话题。如果你并没有太好的想法，那就和我一样使用 RESTful 风格的 API 吧。

4.1.4 RESTful API

REST 是由英文 Representational State Transfer 的缩写，即表征性状态传输，它是为了描述现代 Web 如何工作而提出的。REST 是一种设计风格，不是一种标准。

受用户欢迎的网站，绝大多数都采用了这种风格的设计。REST 依赖于 HTTP 协议的功能，使客户端与服务端间的交互需要遵守特定的协议（或者说是规范），如表 4-2 所示。

表 4-2 REST 规范

HTTP 方法	行为
GET	获取（读取）资源
POST	创建资源（创建下一个）
PUT	更新资源
DELETE	删除资源

设计 RESTful API 的关键便是设计好 URI，即指定资源间的关系。我们一直讨论的资源，你可以将之视为数据——可以存储到计算机中的事物。通常这时候我们会遵循一些默认的规则：

（1）使用斜杠（/）来区分资源间的关系。如blog/1为第一篇博客，blog/1/comment/1应该是第一个博客的第一条评论。

（2）如果我们需要在博客中使用多个单词，如mobile devices，那么我们可以考虑用减号（-）或者下画线（_）来区分它们，即mobile_device或者mobile-device。

（3）额外参数。我们可能只需要返回10个结果的博客，那么我们会在URI的后面加上?limit=10，用来限定返回的数量。如果这时候有多个参数，如?limit=10&offset=10，那么应该是返回10个后面的10个，即第11~20个博客。

如果你用过一些博客系统，如CSDN的博客系统，有这样一个URI: http://blog.csdn.net/ phodal/article/details/17482213。

按照第一条规则，我们可以重新解读一下这条URI。除去域名就是phodal/article/details/17482213，这里的phodal即为笔者的用户名；article指代的就是文章；details说的就是文章的详情页，即文章的详细内容；随后的17482213，可能是在数据库中的id。

现在，让我们来GET一个复杂的API（笔者博客的API：https://www.phodal.com/api/v1/ app/?format=json&limit=10&offset=10）：

GET /api/v1/app/?format=json&limit=10&offset=10

这里的“api”即指定了这是一个API专用的URI。v1说明了API的版本，在一些用途比较广泛的公有API中，都会有类似的版本标识。“app”在这里有一个特别用途，表明这个API专用于手机应用。随后的?format=json&limit=10&offset=10，表明了返回的格式是json，同时限制了数量是10个，偏移是10个，即从第11个开始。

多说无益，让我们来试试。

RESTful API实战

尽管Node.js自身提供了很多模块，然而直接使用这些基本的模块来开发应用需要花费大量的时间。如上面示例的Node.js的HTTP服务，如果使用别的模块，我们

可以使用更简洁的代码。在我们安装 Node.js 的时候，系统会默认安装上 NPM。NPM 全称是 Node Package Manager，它是 Node.js 用于管理软件包的工具，这个工具和我们第 2 章提过的 Mac OS 上的 brew、Windows 上的 choco 以及 Ubuntu 上的 apt-get 有着相同的功能。

Express 是一个简洁、灵活的 Node.js Web 应用开发框架，它可以帮助我们创建各种 Web 应用。在后面的章节里，我们将用 Express 作为 HTTP 协议的基础框架，并在此基础上添加更多的协议和应用。

首先，让我们创建一个 app.js 文件，在里面添加 GET 请求的处理：

```
var express = require('express');
var app = express();

app.get('/api', function(req, res){
    res.send({led: false});
});
```

同理，我们可以添加其他方法，即 post、put、delete。不论它们发过来怎样的方法，都是返回{led: false}，即 JSON API。

```
var express = require('express');
var app = express();

app.get('/api', function(req, res){
    res.send({led: false});
});

app.post('/api', function (req, res) {
    res.send({led: false})
});

app.put('/api', function (req, res) {
    res.send({led: false})
});

app.delete('/api', function (req, res) {
    res.send({led: false})
```

```
});

app.listen(8080);
```

现在，让我们启动这个 HTTP 服务，然后对其运行测试：

```
node app.js
```

GET 方法可以用网页来实现，POST、DELETE、PUT 方法可以用 Curl 或者 Chrome 的插件 Postman。Postman 使用时的截图如图 4-5 所示。

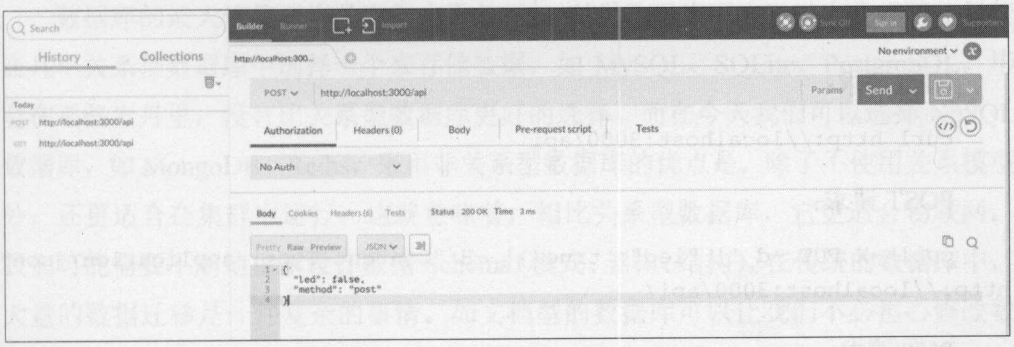


图 4-5 Postman 界面截图

我们向 http://localhost:3000/api 发出了一个 POST 请求，结果它返回了：

```
{
  "led": false,
  "method": "post"
}
```

或许你发现了，这不是上面的代码做的事，而下面的代码才是：

```
...
var data = {led: false};

...

app.post('/api', function (req, res) {
    data.method = 'post';
    if (req.body.led === true) {
        data.led = true;
    } else if (req.body.led === false) {
```

```
            data.led = false;
        }
        res.send(data);
    });

    ...
```

当我们向 API 发送（POST）数据的时候，如果请求的 body 中包含 led 的值，并且其值为 true，那么我们就将 led 的值设置为 true，否则将其设置为 false。

如果你用的是 curl，那么可以用下面的脚本来测试：

GET 请求：

```
curl http://localhost:3000/api
```

POST 请求：

```
curl -X PUT -d '{ "led": true }' -H "Content-Type: application/json"
http://localhost:3000/api/
```

PUT 请求：

```
curl -X POST -d '{ "led": true }' -H "Content-Type: application/json"
http://localhost:3000/api/
```

DELETE 请求：

```
curl -X DELETE http://localhost:3000/api
```

在上面的参数中：

- -X，指定 HTTP 的请求方式。
- -d，传送的数据。
- -H，自定义头信息传递给服务器。

如 `curl -X PUT -d '{ "led": true }' -H "Content-Type: application/ json" http://localhost: 3000/api/`的意思便是将一个数据为{ "led": true }、头为 Content-Type: application/json 的 PUT 请求发送给服务器。

4.2 数据持久化

所谓的数据持久化，便是可以将数据永久地保存在存储设备中。而数据持久化的形式又多种多样，如文件存储、存储在数据库中等，最后都会以文件的形式存储到文件系统中。至于使用哪种存储形式则取决于你的设计。

4.2.1 数据库简介

数据库的最大价值或许是存储大量数据[2]。对于数据库而言，过去我们可能只能在几个关系型数据库中选择一个来存储数据，如 MySQL、SQLite、PostgreSQL。毕竟在那些岁月里，没有比关系型数据库更好的选择。而在今天我们可以选择 NoSQL 数据库，如 MongoDB、Redis，采用非关系型数据库的优点是，除了不使用关系模型外，还更适合在集群中运行。这就意味着，相比关系型数据库，它更适合物联网。我们可能需要不断地重新设计数据 Schema（模式、架构、结构），在传统的数据库中，大量的数据迁移是一件复杂的事情。如文档型的数据库可以让我们不必担心修改数据结构带来的问题。

我们需要知道的是，数据库的种类不止有关系型和文档型数据库两种，还有键-值数据库 Redis、Riak，类型数据库 HBase，图数据库 Neo4j 等。

如果你需要搜索的功能多于存储和读出，那么可以考虑 ElasticSearch。

1. 关系型数据库

关系型数据库，是建立在关系模型基础上的数据库，借助于集合代数等数学概念和方法来处理数据库中的数据。通常这会依据于现实世界的实体，以及它们之间的联系来设计。关系型数据库在过去几十年里，乃至现在都很流行，究其原因我们可以发现：

（1）其符合大多数人的思维方式，概念贴近于逻辑世界。

（2）遵循 ACID 规则，即原子性、一致性、独立性、持久性。

2 来源于 Martin Fowler 的《NoSQL 精粹》。

（3）易于使用和维护。

（4）功能强大，支持复杂的查询、存储过程、高级索引、触发器等。

如果你有过 SQL 的使用经验，那么我想你也有过这样的经历。在最开始的时候，我需要设计一个 Schema，这个 Schema 定义了表、每个表的字段，还有表和字段之间的关系。 如现在，我们来设计一个用户登录的表的字段，如表 4-3 所示。

表 4-3

字段	类型
id	INTEGER
name	STRING
password	STRING
expiration	DATE
isAdmin	BOOLEAN
uid	STRING
phone	STRING
createdAt	DATE
updatedAt	DATE

从表 4-3 中可以看到，我们定义了数据类型，这就意味着我们必须严格按照这些类型来创造数据。假定这个表的名字是 user，那么我们需要查询某个 id 对应的用户名，因而它的 SQL 语句应该就是：

```
SELECT *
FROM user
WHERE id = 1
```

这样就会返回这个用户相关的信息，我们便可以在逻辑层对其进行处理。

我们需要预先想好这些，对于一个自顶向下设计的系统，或者文档驱动的系统来说这不是问题。但是如果是业务驱动的系统可能就会存在问题。

2. NoSQL

由于关系型数据库发明的年代并没有预见到集群的流行。过去由于硬件每 18 个月性能就会增加一倍，因而升级硬件就是一个很明智的选择。但是这时成本会不断

地提高，提升内存的同时，我们还需要提升 CPU，等等。这时，另外一个瓶颈就出现了，即关系型数据库。关系型数据库虽然可以将负载分散到多个服务器，但是这样做需要付出大量的开发应用程序时间。

NoSQL 就在这种情况下诞生了——运行在集群中的数据库。NoSQL 不使用传统的关系数据库模型，不需要预先设计表和结构就可以储存新的数值。从分类上来说，NoSQL 可以分成四大类：

（1）键值（Key-Value）存储数据库。键值数据库以 Key-Value 的形式来存储数据，类似于传统语言的哈希表。常见的键值数据库有 Riak、Redis、LevelDB、Memcached 等，现在越来越多的架构采用这样的数据库作为缓存服务器，如 Redis、Memcached。

（2）列存储数据库。列存储数据库将数据存储在列族（column family）中，通常一个列族存储经常被一起查询的相关数据。像近几年来流行的 Hadoop 就是以 HBase 来提供服务的。

（3）文档型数据库。文档型数据库的数据通常会以文档的形式来存储。在这个类型的数据库中，比较流行的有 MongoDB、CouchDB、RavenDB。

（4）图形（Graph）数据库。图数据库可以以图的方式存储数据。

具体使用哪个数据库，取决于你所需要实现的功能，在这里我们选用的是文档型数据库。

4.2.2 连接 MongoDB 数据库

MongoDB 是一个面向文档的数据库。这就意味着它包含文档数据库的一些特性，如以文档的的形式来存放、读取数据，并且我们可以在文档中嵌入文档和数组。它是一个 JSON 文档数据库，但是在保存的时候会以二进制 JSON 的形式保存，即 BSON。当然，MongoDB 还支持传统数据库的创建、读取、更新和删除数据的功能。除此，它也支持索引、聚合、特殊的集合类型等。

其实，我们决定在这里使用 MongoDB 的一个重要原因是：横向扩展。在设计系统的时候，我们需要高伸缩性的数据存储。MongoDB 可以通过横向扩展带来强大的

集群性能，这就意味着我们只需将我们的精力集中于编写应用程序上，而不需要考虑系统的扩展问题。当集群需要处理更多的数据的时候，我们只需添加新的服务器，而不是购买一个更好的服务器来替换旧的服务器，单个系统的硬件都存在极限。在笔者经历过的一个项目中，系统用搜索引擎来存储数据，由于搜索引擎所能存储的数据将要达到硬件极限（512GB 内存），即已经无法通过扩展硬件来存储数据，这时我们就选择重构系统，改用一个支持扩展的搜索引擎。重构现有的系统不仅没有多少商业价值，而且会花费大量的宝贵时间。

在本书中，只会讨论基本的 CRUD，即创建、读取、更新和删除。现在，让我们来安装一个 MongoDB，并在我们的程序中使用它。

1. 安装 MongoDB

由于我们没有办法限制用户所使用的环境，所以在这里我们还是提供不同平台的安装方法。

Windows 下可以到官网下载: https://www.mongodb.org/downloads，或者用前文中说到的 **Chocolatey** 安装：

```
choco install mongodb
```

Ubuntu 等下也是类似的（在有些情况下，你可能需要安装软件源等，建议可以参照官方的安装指南：https://docs.mongodb.org/manual/administration/install-on-linux/）：

```
sudo apt-get install -y mongodb-org
```

Mac OS 下，可以执行：

```
brew install mongodb
```

安装完成后，依据相关的系统启动 MongoDB Server。如图 4-6 所示是在 Mac OS 上启动 MongoDB 时的 log。

```
fdhuang ~/Desktop > mongod
mongod --help for help and startup options
2015-10-26T21:49:56.469+0800 [initandlisten] MongoDB starting : pid=86459 port=27017 dbpath=/data/db 64-bit host=phodal
2015-10-26T21:49:56.469+0800 [initandlisten]
2015-10-26T21:49:56.469+0800 [initandlisten] ** WARNING: soft rlimits too low. Number of files is 256, should be at least 1000
2015-10-26T21:49:56.469+0800 [initandlisten] db version v2.6.5
2015-10-26T21:49:56.469+0800 [initandlisten] git version: nogitversion
2015-10-26T21:49:56.469+0800 [initandlisten] build info: Darwin phodal.local 14.0.0 Darwin Kernel Version 14.0.0: Fri Sep 19 00:26:44 PDT 2014; roo
t:xnu-2782.1.97~2/RELEASE_X86_64 x86_64 BOOST_LIB_VERSION=1_49
2015-10-26T21:49:56.469+0800 [initandlisten] allocator: tcmalloc
2015-10-26T21:49:56.469+0800 [initandlisten] options: {}
2015-10-26T21:49:56.470+0800 [initandlisten] journal dir=/data/db/journal
2015-10-26T21:49:56.470+0800 [initandlisten] recover : no journal files present, no recovery needed
2015-10-26T21:49:56.511+0800 [initandlisten] waiting for connections on port 27017
^C2015-10-26T21:49:58.270+0800 [signalProcessingThread] got signal 2 (Interrupt: 2), will terminate after current cmd ends
2015-10-26T21:49:58.271+0800 [signalProcessingThread] now exiting
2015-10-26T21:49:58.271+0800 [signalProcessingThread] dbexit:
2015-10-26T21:49:58.271+0800 [signalProcessingThread] shutdown: going to close listening sockets...
2015-10-26T21:49:58.271+0800 [signalProcessingThread] closing listening socket: 8
2015-10-26T21:49:58.271+0800 [signalProcessingThread] closing listening socket: 9
```

图 4-6 MongoDB 启动过程

在这里，我们还需要安装 Node.js 的 MongoDB 模块，可以通过下面的代码来安装

```
npm install mongodb --save
```

过程如图 4-7 所示。

```
fdhuang ~/Desktop/designiot ‹ master › ~/designiot > npm install mongodb --save

npm WARN optional dep failed, continuing kerberos@~0.0
mongodb@2.0.46 node_modules/mongodb
├── es6-promise@2.1.1
├── readable-stream@1.0.31 (isarray@0.0.1, inherits@2.0.1, string_decoder@0.10.31, core-util-is@1.0.1)
└── mongodb-core@1.2.19 (bson@0.4.19)
```

图 4-7 安装 Node.js 的 MongoDB 模块

接着，我们就可以写一个简单的程序来连接数据库了。

```
var MongoClient = require('mongodb').MongoClient;

var url = 'mongodb://localhost:27017/designiot';
MongoClient.connect(url, function(err, db) {
  console.log("Connected correctly to server");
  db.close();
});
```

首先，我们引入了 MongoClient 模块。接着，url 指向了本地 MongoDB 的地址，url 中的 designiot 是数据库名。然后，我们连接了数据库，当连接成功的时候，就会在控制台上打出 log。最后，将关闭数据库。我们连接到 MongoDB 时的日志如图 4-8 所示。

```
2015-10-26T22:05:00.111+0800 [initandlisten] connection accepted from 127.0.0.1:56684 #17 (5 connections now open)
2015-10-26T22:05:00.120+0800 [conn17] end connection 127.0.0.1:56684 (4 connections now open)
2015-10-26T22:05:00.125+0800 [initandlisten] connection accepted from 127.0.0.1:56685 #18 (5 connections now open)
2015-10-26T22:05:00.126+0800 [initandlisten] connection accepted from 127.0.0.1:56686 #19 (6 connections now open)
2015-10-26T22:05:00.126+0800 [initandlisten] connection accepted from 127.0.0.1:56687 #20 (7 connections now open)
2015-10-26T22:05:00.129+0800 [initandlisten] connection accepted from 127.0.0.1:56688 #21 (8 connections now open)
2015-10-26T22:05:00.129+0800 [initandlisten] connection accepted from 127.0.0.1:56689 #22 (9 connections now open)
2015-10-26T22:05:00.132+0800 [conn18] end connection 127.0.0.1:56685 (8 connections now open)
2015-10-26T22:05:00.132+0800 [conn19] end connection 127.0.0.1:56686 (7 connections now open)
2015-10-26T22:05:00.132+0800 [conn20] end connection 127.0.0.1:56687 (7 connections now open)
2015-10-26T22:05:00.132+0800 [conn21] end connection 127.0.0.1:56688 (6 connections now open)
2015-10-26T22:05:00.132+0800 [conn22] end connection 127.0.0.1:56689 (6 connections now open)
```

图 4-8　连接到 MongoDB

在完成连接数据库后，我们就可以重写我们在第 2 章中提到的 API 了。

2. 操作数据库

我们需要的是类似于表 4-4 所示的 API。

表 4-4　我们需要的 API

URL	结果
/api	返回所有结果
GET /api/:user_id	返回指定 user_id 的结果
POST /api/:user_id	向指定 user_id 发送/更新数据
PUT /api/:user_id	向指定 user_id 发送/更新数据
DELETE /api/:user_id	删除指定 user_id 的数据

现在，让我们来创建一个 db.js 的文件，并写一个插入数据的方法：

```
var MongoClient = require('mongodb').MongoClient;
var url = "mongodb://localhost:27017/designiot";

function MongoPersistence() {
}

MongoPersistence.prototype.insert = function (payload) {
    'use strict';
    MongoClient.connect(url, function (err, db) {
        var insertDocuments = function (db, callback) {
            var collection = db.collection("documents");
            collection.insert(payload, function (err, result) {
                callback(result);
            });
```

```
        };
        insertDocuments(db, function () {
            db.close();
        });
    });
};
```

我们创建了一个名为 MongoPersistence 的方法，顾名思义就是专门用于 MongoDB 的数据持久化。接着，我们在这个原型上创建了一个名为 insert 的方法，用于插入数据。然后在连接上数据库的时候，创建了一个叫作 insertDocuments 的方法，在这个方法里面执行相关的数据操作。最后，关闭数据库。

在这里，我们需要知道 collection 的概念。MongoDB 中的集合（collection）的概念类似于关系型数据库中表（table）的概念，故而在此一个文档就相当于表中的一行。我们可以对两种类型的数据库做一个简单的对比，如表 4-5 所示。

表 4-5 两种数据库的对比

MySQL	MongoDB
模式（schema）	数据库（database）
表（table）	集合（collection）
行（row）	文档（document）

collection 的 insert 函数，可以将一个文档添加到集合中。只要我们的 payload 是有效的 MongoDB 文档，即满足 JSON 格式，就可以保存到 documents 集合中。

随后，我们可以在 app.js 文件中，导入 db.js 库，并修改 app.post 中的回调函数。当我们进行 PUT 请求的时候，直接向数据库插入相关的数据。

```
var Database = require('./db');
var db = new Database();
...
app.put('/api', function (req, res) {
    var data = {led: false};
    if (req.body.led === true) {
        data.led = true;
    }
```

```
        db.insert(data);
        res.send({db: "insert"});
    });
```

当然，最后我们需要返回是否成功插入数据。由于上面的数据的内容只有led=true，很难区分两个不同的 led，以及两个不同的数据。因此建议将上面的 data 改成一个定值，一个为 led 的数据，如：

```
var data = {led: true};
```

随后，我们可以在 MongoDB 的控制台，查询这个数据，如图 4-9 所示。

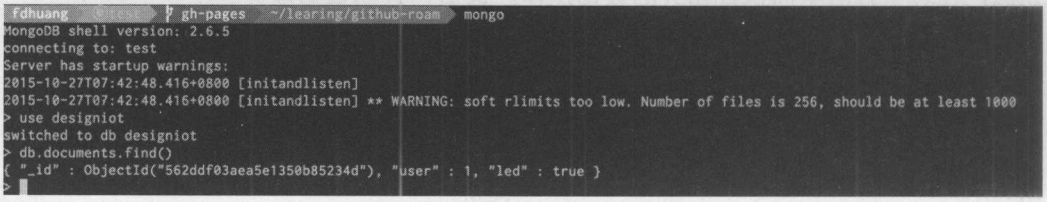


图 4-9　MongoDB 查询数据

我们需要输入 mongo 进行 MongoDB shell，接着用 use designiot 切换到 designiot 这个数据库。MongoDB 查询数据的语法格式如下：

```
db.COLLECTION_ NAME.find()
```

我们只需将上面的COLLECTION_NAME换成**designiot**，就可以有下面的结果：

```
{ "_id" : ObjectId("562ddf03aea5e1350b85234d"), "user" : 1, "led" : true }
```

这里的_id 是自动生成的。这个 ObjectId 由时间戳、机器进程 ID 和自增计数器组成。现在，我们就需要从数据库中取出这个值，再放到前端显示。

同上面的 insert 方法，我们可以用 MongoDB 中的 find 方法来查找我们的数据。下面是我们最后写成的查找数据的代码：

```
MongoPersistence.prototype.find = function (queryOptions, queryCB) {
    'use strict';
    MongoClient.connect(url, function (err, db) {
        var findDocuments = function (db, query, callback) {
            var collection = db.collection("documents");
            collection.find(query).toArray(function (err, docs) {
```

```
            callback(docs);
        });
    };

    findDocuments(db, queryOptions, function (result) {
        db.close();
        queryCB(result);
    });
  });
};
```

insert 与 find 的主要不同在于下面的代码，在查找完成后，将数据转换为数组。

```
collection.find(query).toArray(function (err, docs) {
  callback(docs);
});
```

在完成查询后，用一个回调函数 queryCB 来返回这个结果。

现在，我们就可以修改我们的 app.get 中的回调函数了：

```
app.get('/api', function (req, res) {
  var payload = {user: 1};
  db.find(payload, function(results){
    return res.json(results);
  });
});
```

当我们请求/api 这个 URL 的时候，我们会在数据库中查询 user=1 的相关数据并返回所有结果。

3. 更新数据

在插入了多个数据后，我们发现每次都返回了多个结果，这不是我们想要的。对于控制一个 LED 来说，我们只需要一个状态。这就意味着，首先需要找到之前的数据，然后更新该数据。因此，我们需要再次更新应用，让它做更多的事：

```
app.put('/api', function (req, res) {
  var payload = {user: 1};
  var data = {user: 1, led: false};
```

```
    if (req.body.led === true) {
        data.led = true;
    }

    db.find(payload, function(results){
        if(results.length>0){
            db.update(data);
            res.send({db: "update"});
        } else {
            db.insert(data);
            res.send({db: "insert"});
        }
    });
});
```

首先，我们会在数据库中查找 user=1 的相关数据，如果有相关的数据，那么我们就更新这个数据，如果没有这个数据那么我们就插入它。相应的，我们就需要在 db.js 中有一个 update 函数：

```
MongoPersistence.prototype.update = function (payload) {
    'use strict';
    MongoClient.connect(url, function (err, db) {
        var updateDocument = function (db, callback) {
            var collection = db.collection("documents");
            collection.update({user: payload.user}, {$set: {led:
payload.led}}, function (err, result) {
                callback();
            });
        };
        updateDocument(db, function () {
            db.close();
        });
    });
};
```

update 方法与 insert 的区别，还是在集合的操作部位。首先，数据库会去查询要更新的部分，这里的{user: payload.user}就是一个 query。接着，第二部分的{$set: {led: payload.led}}是我们要操作的部分，在这里我们只需更新 led 的状态。最后，返回运

行回调函数，并结束内容。

在完成了查询和更新之后，我们还需要删除数据，这就留给读者去实现了。因为通常我们很少会去删除数据，所以在这里就不实现这个功能了。我们还需要添加一个/api 的路由，用于返回所有结果：

```
app.get('/api/', function (req, res) {
    var payload = {};
    db.find(payload, function(results){
        return res.json(results);
    });
});
```

与 find 方法不同的是，这里的 payload 是{}，即空对象。这样 MongoDB 便会查询并返回所有的结果。

4. 多用户

我们花费了很大经历来构建我们的 API，但是当前仅支持一个设备，现在我们需要支持多个设备。有了上面的代码后，修改起来就相当简单了，最后代码如下：

```
var express = require('express');
var bodyParser = require('body-parser');
var Database = require('./db');
var db = new Database();

var app = express();
app.use(bodyParser.json());

app.get('/api/', function (req, res) {
    var payload = {};
    db.find(payload, function(results){
        return res.json(results);
    });
});

app.get('/api/:user_id', function (req, res) {
    var payload = {user: req.params.user_id};
    db.find(payload, function(results){
```

```
        return res.json(results);
    });
});

function updateData(req, res) {
    var payload = {user: req.params.user_id};
    var data = {user: req.params.user_id, led: false};
    if (req.body.led === true) {
        data.led = true;
    }

    db.find(payload, function (results) {
        if (results.length > 0) {
            db.update(data);
            res.send({db: "update"});
        } else {
            db.insert(data);
            res.send({db: "insert"});
        }
    });
}

app.post('/api/:user_id', function (req, res) {
    updateData(req, res);
});

app.put('/api/:user_id', function (req, res) {
    updateData(req, res);
});

app.delete('/api/:user_id', function (req, res) {
    res.send({});
});

app.listen(3000);
```

我们所做的事情就是用/api/:user_id 替换了/api 这个路由，并用 req.params.user_id 来替换上面代码中的 1。最后的代码位于代码库的第 4 章的 rest-db 目录中。

现在，让我们来测试一下。我们将 user 1 的 led 置为 true，将 user 2 的 led 置为 false。

```
curl -X PUT -d '{ "led": true }' -H "Content-Type: application/json"
http://localhost:3000/api/1
curl -X PUT -d '{ "led": false }' -H "Content-Type: application/json"
http://localhost:3000/api/2
```

最后，我们将得到如图 4-10 所示的结果。

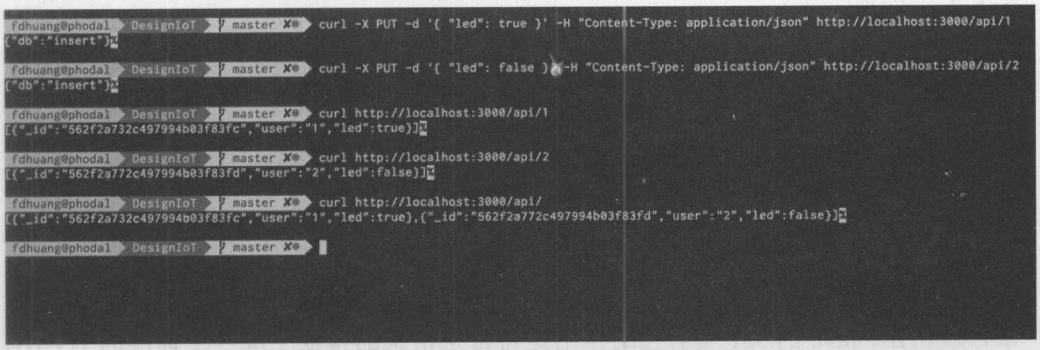


图 4-10　获取 API 结果

当我们请求 http://localhost:3000/api/1 时，会返回 user 1 相关的结果。当我们请求 http://localhost:3000/api/2 时，会返回 user 2 相关的结果。当我们请求 http://localhost:3000/ api/时，会返回所有的结果。

而根据不同的结果，我们会在这里面存储不同用户的不同数据。使用 NoSQL 的一个优势便在于此，我们不需要考虑用户的数据是什么，只需校验这个数据的数据格式是不是正确。如果格式正确，则在用户请求这些数据的时候，返回同样的数据。

这样一个基础的 RESTful API 就可以帮助我们完成程序基本框架，只需在上面加上相关的业务逻辑，如数据处理、用户授权等。由于业务是程序的核心，而不同的人又会有不同的业务需求，因此这里不再一一展开。但是，在后面的章节中，我们会构建一个支持多协议的应用。

4.3 视图与应用层

实际上，今天我们可以将这个层级分成多个部分。幸好最开始的时候，我们对系统已经有一个很好的概念，我们需要支持不同的协议，以支持不同的平台，如移动应用和微信的难度并不是很大。但是，在那之前我们需要一个图形用户界面，这样我们的用户就可以在浏览器上操作他们的设备了。

4.3.1 视图

值得注意的是，如果你只希望用户在手机上控制设备，那么视图层可能就不是必需的。一般来说，在这一层不会存在、也不应该存在复杂的逻辑，它应该只用来显示数据。

在浏览器端的视图层，最后的形式就是 JavaScript + HTML + CSS。而通常来说，HTML 是最后展现的形式，并非在代码中的形式。在不同语言下会有不同的模板引擎，如 PHP、Java 的 JSP 等，它的作用是**将特定格式的模板生成一个标准的 HTML 文档**。

下面是一个 HTML 的“Hello,World”示例：

```
<html>
 <head>
  <title>PHP Test</title>
 </head>
 <body>
 <p>Hello World</p>
 </body>
</html>
```

而上面的代码是经过 PHP 的模板引擎处理完后的结果。其原来的 PHP 代码是：

```
<html>
 <head>
  <title>PHP Test</title>
 </head>
 <body>
 <?php echo '<p>Hello World</p>'; ?>
```

```
 </body>
</html>
```

在原来的PHP代码里，我们只是做了一个echo，即PHP中的输出。在多数情况下，我们可以将之视为对字符串的处理。一个好的模板引擎，可以加快我们的开发速度，并提高代码的可读性。例如，我们将会使用jade，它是一个高性能的简洁模板引擎。如果你有写过很多XML，那么你可能已经讨厌 <html></html>这样的形式。在jade中，你可以这样表达：

```
html
  head
    title my jade template
  body
    h1 Hello World
```

随后，上面的代码会被转译成：

```
<html>
  <head>
    <title>my jade template</title>
  </head>
  <body>
    <h1>Hello Bob</h1>
  </body>
</html>
```

我们只需在相应的层级中添加一个缩进与HTML标签，便可以形成上面的代码。

在Android、iOS或Windows Phone上开发界面的时候，都有相应的一些View组件。我们看到的多数应用的按钮、菜单等都是以组件化的形式出现的。在浏览器上虽然也有类似的组件，但是由于不同浏览器之间存在兼容性差异，因此我们需要考虑使用一些前端框架。

通常在一些比较大的项目里会有专门的人员来维护界面，或者说是前端，他们可能会自己去创造框架来满足自己的需求。对于一般的团队来说，这时候最好的选择是使用已有的框架来设计自己的视图层。

现在，我们创建一个简单的控制界面。

4.3.2 控制层界面

1. 基本的界面

为了在 Express 中使用模板引擎，我们需要名为 Jade 的模块。这是 Express 官方默认的模板引擎，并且是 Node.js 官方推荐的。在上面的例子中，可以看到它的写法比较简单和优雅。现在，我们先安装这个模板引擎：

```
npm install jade --save
```

首先，我们需要修改 app.js，先添加一个新的 body 解析器，用于解析表单请求中的数据。

```
app.use(bodyParser.urlencoded({extended: true}));
```

然后，我们需要设置模板引擎及其目录，即 path.join 会加入当前项目的路径下的 views 文件夹中的所有模板文件。

```
app.set('views', path.join(__dirname + '/', 'views'));
app.set('view engine', 'jade');
```

接着，创建一个路由来指定使用的模板：

```
app.get('/', function (req, res) {
    'use strict';
    res.render('index', {
        title: 'Home'
    });
});
```

这里的 res.render('index',{title: 'Home'})中的 index 指代的是 index.jade 文件，而后面的{title: 'Home'}则是传入的对象，title 相当于 key，后面的 Home 是 value。这个变量将传给模板引擎，再由模板引擎渲染出页面。

最后，我们需要创建我们的模板文件 index.jade：

```
doctype html
html
```

```
head
    title=title
body
    h1=title
```

Jade 使用=来输出一个变量，这些传给 Jade 模板的数据称之为 locals，如这里的 Home。这时候我们的项目目录结构应该如下所示：

```
.
|____app.js
|____db.js
|____package.json
|____views
| |____index.jade
```

现在，让我们运行服务器，看看效果怎么样？如图 4-11 所示。

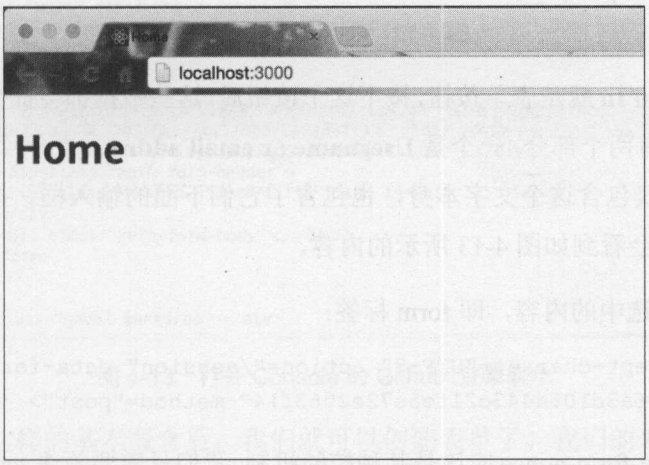


图 4-11　Jade 模板引擎的 Hello,World

2. 创建控制界面

在完成一个很简单的“Hello,World”后，我们就可以尝试在网页上控制我们的硬件。你可能使用过一些比较高级的技术，如 Ajax，可以在我们不刷新页面的情况下修改结果。但是在这里，我们还是以表单为例来做一个简单的控制界面。

表单在 Web 中的用处是**允许用户输入数据，并将这些数据发送给服务器**。通常

来说表单会由三部分组成：form 标签、表单域和表单按钮，但是对于用户可见的只有后两者。如图 4-12 所示的是 Github 的登录表单。

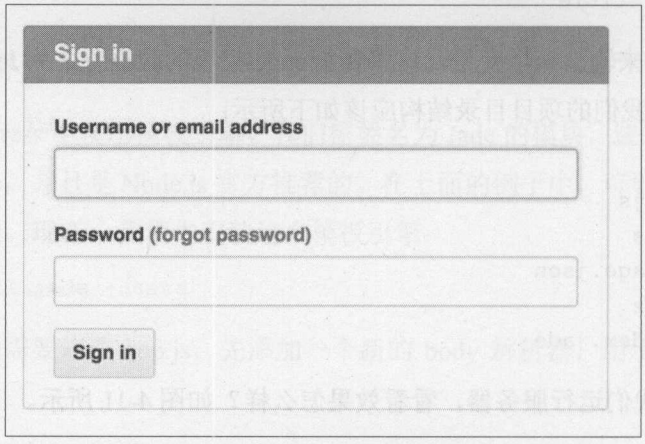


图 4-12　Github 的登录表单

这里的 **Sign In** 就是表单按钮，按下这个按钮后，这些数据都会提交到服务器端。这里的表单域有两个部分：一个是 **Username or email address**，另一个是 **Password**，这里的内容不仅包含这个文字本身，也包含了它们下面的输入框。打开这个页面的 Console，我们会看到如图 4-13 所示的内容。

图 4-13 中选中的内容，即 form 标签：

```
<form accept-charset="UTF-8" action="/session" data-form-nonce=
"1329a624a2406a3d106a443c21fe5e72e2063f14" method="post">
```

这里的 data-form-nonce 应该是某种校验机制，我们只需要关注 action 和 method。这里的 action 用于指定处理提交表单的格式，在这里是 URL/session。而 method 则是指出了表格提交的方式应该是 POST 请求。

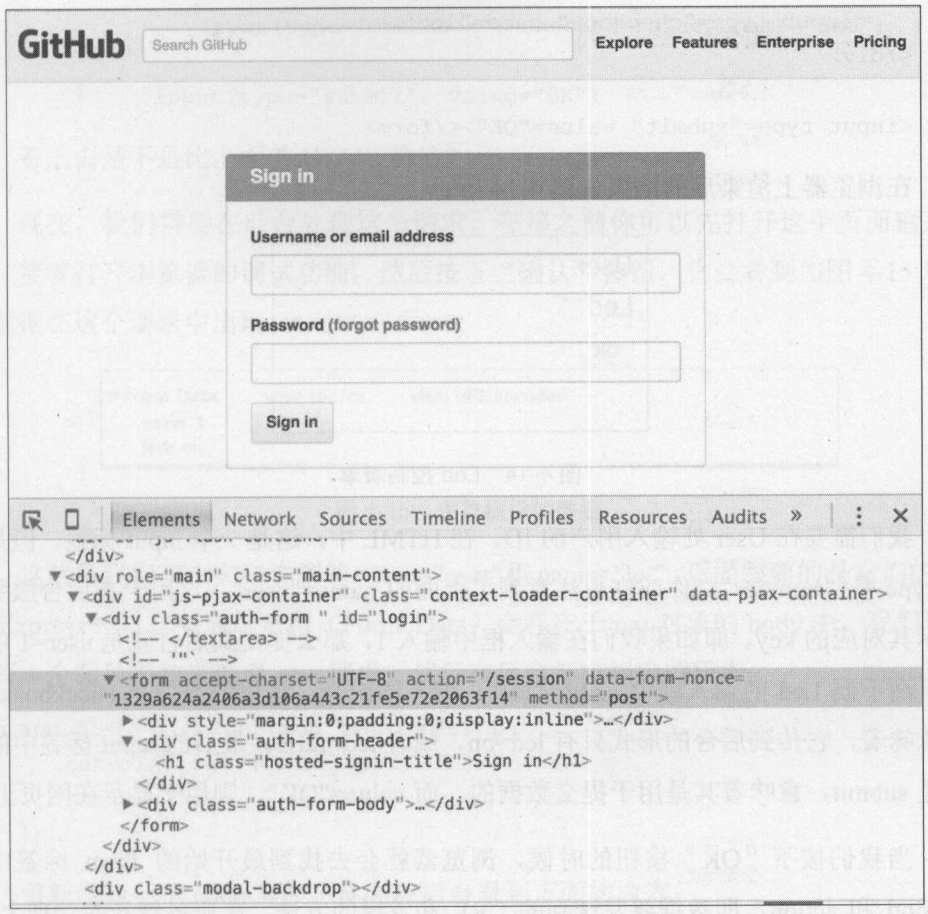


图 4-13 打开 Console 的 Github 登录表单

在理解了这样的基本概念后，我们就可以创建表单了。我们的表单域有两个，一个是 user 的 id，另一个是 led 的状态。下面是我们最后想要的 HTML 代码：

```
<form method="post" action="/">

<div>
   <label from="user">User</label>
   <input type="number" name="user">
</div>

<div>
   <label from="led">Led</label>
```

```
    <input type="checkbox" name="led">
</div>

<input type="submit" value="OK"></form>
```

在浏览器上渲染后的结果如图 4-14 所示。

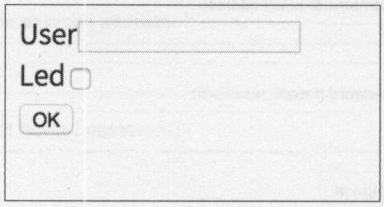


图 4-14　Led 控制表单

我们需要在 User 处输入用户的 ID。在 HTML 中，这是一个 input 标签，但是类型 type 是“number”。你或许也注意到了上面的 name="user"，这用于在后台服务器显示其对应的 key。即如果我们在输入框中输入 1，那么提交到后台就是 user=1 的形式。而下面 Led 的输入框中 name="led"，但不同的是，这里的类型是 checkbox。这就意味着，它传到后台的形式只有 led=on，或者 led=false。最后的 input 标签中的类型是 submit，意味着其是用于提交数据的。而 value="OK"，则用于显示在网页上。

当我们按下“OK”按钮的时候，浏览器就会去找到最开始的 form 标签中的 method 和 action，即数据要发送到的 URL 和发送的方法。在服务器开始处理之前，我们需要先将上面的代码转为 Jade 模板的形式：

```
doctype html
html
    head
        title= title
    body
        form(method="post", action="/")
            div
                label(from="user") User
                input(type='number', name="user")
            div
                label(from="led") Led
```

```
        input(type='checkbox', checked=false, name="led")

    input(type="submit", value="OK")
```

看上去是不是比上面的 HTML 简洁？

现在，我们需要在后台处理这个请求。在那之前你可以先打开这个页面输入数据，接着打开浏览器的调试功能，然后按下“确认”按钮，你会看到如图 4-15 所示的数据在这个请求中出现。

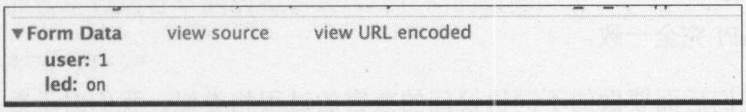


图 4-15 表单提供的数据

这就是我们在上文中说到的 name="user"和 name="led"，后面跟随的是它们的值。在 Express 中，这些表彰数据（Form Data）会存在于 req 对象的 body 中。我们可以先写一个简单函数来处理 post 请求，然后在后台打出相应的日志：

```
app.post('/', function (req, res) {
    console.log(req.body);
    res.end();
});
```

重新运行一下 node 程序，你会在后台看到下面的内容：

```
{ user: '2', led: 'on' }
```

随后，我们所要做的事，和之前 API 做的事情是类似的：

```
app.post('/', function (req, res) {
    var userID = req.body.user;

    var payload = { user: userID};
    var led = req.body.led === "on";

    var data = { user: userID, led: led};
    db.find(payload, function (results) {
        if (results.length > 0) {
            db.update(data);
```

```
            res.send({db: "update"});
        } else {
            db.insert(data);
            res.send({db: "insert"});
        }
    });
});
```

后面的 db.find 的代码甚至是重复的，唯一不同的是，对请求的处理。首先，我们从 req.body.user 中拿到了 user 的 ID，然后判断 req.body.led 是不是等于 on，最后的过程和 API 完全一致。

这和我们后面要做的不同协议间的兼容的过程相类似。我们对于数据的处理是同样的过程，而对于不同协议所做的仅是一个适配。

现在，你再提交一下表单，就可以在后台（即打开/api）看到相应的数据生成了。

4.4 部署

下面，我们试着将我们的应用部署到服务器上。当然你也不一定要有一个真实的服务器，你可以拿你的电脑，或者是你的 Raspberry Pi。如果你有一个 root 过的 Android 手机，也可以试着在上面运行 Debian armf，然后你就可以在上面部署你的应用了。当我们在线上服务器部署 Node.js 项目的时候，会发现要管理 Node.js 的进程并不是一件容易的事。

在我刚开始租用一个服务器的时候，通常在需要部署时我会这样做:

```
git pull --rebase
nohup node app.js &
```

随后开始在项目中写一个重启应用的脚本，这个脚本和一般的自动部署脚本没什么区别。脚本的好处在于减少出错的概率，加快部署速度。但唯一有区别的是，我们需要手动登录这个服务器，然后到项目的目录中去执行脚本。并且在这时候会存在进程挂掉的风险，我们需要一个 Node.js 进程守护模块。

在 Node.js 世界里，有两个进程守护模块比较流行，一个是 forever，另外一个是 pm2。由于 pm2 更适合网站访问量比较大、需要完整地监控界面的系统。在这里我们就简单地介绍一下 pm2，这是一个内建负载均衡的 Node.js 应用的产品级的进程管理器。pm2 可以支持热重启、CPU/内存监视、记录日志、生成启动脚本等。

pm2 本身就是一个 Node.js 模块，我们可以直接用 npm 安装这个模块：

```
npm install pm2 --save
```

接着，我们可以运行下面的脚本来启动我们的服务：

```
pm2 start app.js
```

我们通常用 list、kill 等方法来管理这些进程。

这里我们还需要一个 Nginx 服务器用于反向代理和处理静态资源，同时也可以阻止一些常见的攻击等。而在这时，Nginx 的配置反而比较简单。在这里我们使用的域名是 http://lan.designiot.cn，域名需要和第三方机构申请和购买，并且是以年为单位的。Nginx 会监听那些访问 http://lan.designiot.cn 的 80 端口的请求，并将这些请求定向到 http://127.0.0.1:3000，即我们服务器的端口：

```
server {
    listen 80;
    server_name lan.designiot.cn;

    location / {
        proxy_pass http://127.0.0.1:3000;
    }
}
```

现在，让我们启动 Nginx 服务器，然后就可以添加硬件支持了。

4.5 小结

回顾一下本章开始的图 4-1，我们就能理解客户端/服务端的架构模式了。

在本章中，我们细心雕刻了一个完整的物联网系统。我们不仅可以在页面上控

制 LED 的状态，还可以支持多个 LED 的控制，同时我们还能在 API 级别上对它们进行处理。尽管这是一个非常复杂的理论，但是它实践起来并没有想象中那么难。

4.6 练习建议

（1）使用源码搭建本章的物联网系统。

（2）自己动手敲入本章的代码，并运行。

4.7 相关阅读资料

[1] Leonard Richardson，Mike Amundsen，Sam Ruby. RESTful Web APIs [M]. New York：O'Reilly Media，2013.

[2] [美]Jim Webber，Savas Parastatidis，Ian Robinson. REST 实战[M]. 李锟，俞黎敏，马钧，崔毅译. 南京：东南大学出版社，2011.

[3] [英]Martin Fowler. 企业应用架构模式[M]. 王怀民，周斌译. 北京：机械工业出版社，2004.

[4] [美]Kristina Chodorow，Michael Dirolf. MongoDB 权威指南[M]. 程显峰译. 北京：人民邮电出版社，2011.

[5] [美] Ethan Brown. Node 与 Express 开发[M]. 吴海星，苏文译. 北京：人民邮电出版社，2015.

第5章

连接设备

本章内容

- 连接控制器来控制硬件
- 连接传感器来获取数据
- 连接执行器来执行操作

在本章中我们将简要地介绍一些基本的设备，以及如何连接它们来构成我们的物联网系统。

在第 2 章中，我们说到一个物联网中的物由三个部分组成：控制器、执行器和传感器。对比于计算机，执行器就相当于输出设备，传感器就相当于输入设备，控制器就是计算机。控制器需要负责读取输入设备，并且输出到输出设备。

本章要实现的架构图很简单，仅仅只是从服务器获取状态，并将相应的控制指令发送至执行器，如图 5-1 所示。

本章所需的硬件清单：

- Raspberry Pi 或同等的带 Linux 操作系统的硬件。
- Arduino 开发板与网络模块或者 NodeMCU。

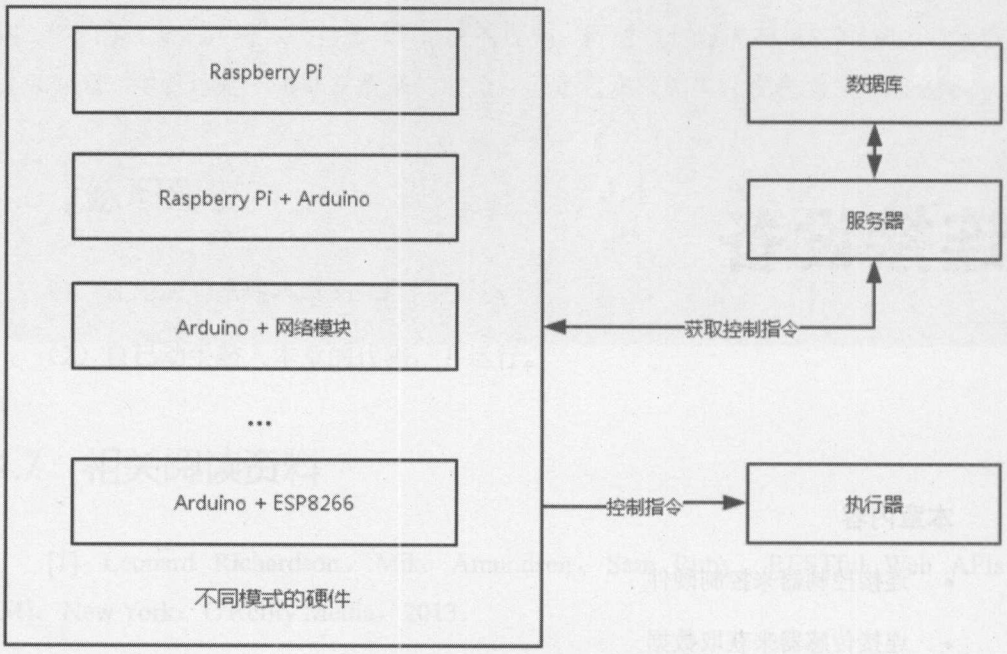


图 5-1　第 5 章程序架构图 1

5.1　连接控制器

现在我们可以尝试连接我们的硬件到服务器上。

由于硬件本身种类繁多——基于同一个芯片有不同的开发板，基于一个开发板会有不同连接方式。在这里将会展示几个不同的开发板及连接方式，以及它们如何连接到我们的服务器上：

（1）Raspberry Pi。直接使用 Raspberry Pi 这类设备的原因是它们有强大的处理能力，并且可以连接传感器、控制硬件。

（2）Raspberry Pi + Arduino。Raspberry Pi 负责了协调层应该做的事情，Arduino 做了硬件层应该做的事情。

（3）Arduino + 网络模块。看似这是一个不错的方案，但是它会受限于 Arduino

的处理器性能，并且会影响后期的扩展性。

（4）NodeMCU。可以将之看成 Arduino + 网络模块的升级版，不仅有了更强大的处理能力，同时成本更低。

而实际上这四种连接方式可以总结为三种连接方案：

（1）控制器负责网络连接和硬件控制。

（2）控制器 1 负责网络连接，控制器 2 负责硬件控制。

（3）控制器负责硬件扩展，并且使用扩展的网络模块。

这里的第一个示例还是使用 Raspberry Pi。这和第 2 章中示例的 API 没有多大区别，我们只想引入一个 Mock API 的概念。

5.1.1 一个重复的示例以及仿造 API

在我们已经有服务端的 API 和代码的情况下，我们并不需要使用这样一个仿造的 API。但是如果我们的服务端代码不是由我们自己所在的团队开发的——依赖于第三方，我们就需要这样一个模仿的 API。由于我们的开发期限所限，我们不能坐着等待这个 API 完成后再进行集成测试。如果你工作于一个传统工作流程的团队，你可能会事先定义好这样一个 API，然后不同的团队分开去做不同的事情。如果你工作于一个敏捷团队，那么这个 API 可能就会用于测试用途。

假定在这时候你们的服务端 API 还没有完成，你会怎么做？

仿造 API

我们可以在 Github（如果你的代码用于生产环境，使用 Github 会带来极大的风险）上创建一个简单的文件来作为 API，也是我们之前提到过的 api.json：

```
{
  "led": true
}
```

为了更表意，将其命名为 led.json，然后将代码托管在 Github 上。先做一些简单

的配置，我们现在就通过 URL——http://api.designiot.cn/led.json 直接访问这个 API。

这时，我们唯一需要做的就是修改 URL，将代码变成：

```
import urllib2,json
import RPi.GPIO as GPIO

GPIO.setmode(GPIO.BOARD)
GPIO.setup(5, GPIO.OUT)

while 1:
    results = urllib2.urlopen('http://api.designiot.cn/led.json').read()
    status = json.loads(results)['led']
    if status == True:
        GPIO.output(5, GPIO.HIGH)
    else:
        GPIO.output(5, GPIO.LOW)
```

当我们需要一个新的接口的时候，我们可能没有办法立即保证这个功能可以完成。这时我们可以做的事情，就是更新一下这个 Mock API：

```
{
  "led": true,
  "lamp": false
}
```

随后我们便可以继续硬件端的开发，而不需要等服务端的开发人员完成他们的工作。

尽管 Raspberry Pi 可以完成这么多功能，但是我们不应该用它完成硬件端的功能。功能强大的设备都可以充当一个很好的协调层。

5.1.2 Raspberry Pi + Arduino

串口通信是指外设与计算机之间，通过数据信号线、地线、控制线等，按位进行传输数据的一种通信方式。如果你接触过单片机编程，想必也对这个通信方式很熟悉。Arduino 与 Raspberry Pi 的串口通信方式有两种，一种是通过 USB 口，另一种是以 GPIO 的形式进行的。在这里我们以 USB 口进行通信，一个是因为其比 GPIO

通信高效，另一个是因为我们可以用这个 USB 口给 Arduino 供电。

在串行通信中，收发双方对发送或接收的数据会按照一定的规则来打包、解包数据，这里最重要的参数有波特率、数据位、停止位和奇偶校验。我们对通信比较关心的是波特率，在某种意义上来说它是数据传输的速率——每秒能传输的数据位。在这里，我们选用 9600 波特率，即每秒能传输 9600 位。需要注意的是，选用较高的波特率，可能会存在一些数据添加或者丢失问题。

如果你已经在计算机上安装了 Python，也可以直接用你的计算机代替 Raspberry Pi 来进行这里的内容。如果你没有安装 Python 环境，可以到 Python 的官方网站下载：https://www.python.org/，或者使用同 Nginx 类似的方式来安装 Python。

在计算机上验证你的代码可以工作后，就可以直接在 Raspberry Pi 上运行相同的脚本了。这也是动态语言的好处，编写即可运行。这里我们可以借助于 Python 的 serial 模块来帮助我们开发。首先，我们需要安装这个模块：

```
pip install serial
```

在一些操作系统（类 UNIX，如 Mac OS、Ubuntu）上，你需要在最前面加 sudo——即超级用户权限，因为这类库是全局使用的。而且，由于这类系统上串口通信属于高级功能，在运行的时候也需要超级用户权限。

如果你在过程中遇到如图 5-2 所示的错误，那么说明你需要用超级用户来进行相关的操作。

```
fdhuang ©test ~ python
Python 2.7.10 (default, Jul 13 2015, 12:05:58)
[GCC 4.2.1 Compatible Apple LLVM 6.1.0 (clang-602.0.53)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import serial
>>> ser=serial.Serial("/dev/cu.usbmodem1451",9600)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/local/lib/python2.7/site-packages/serial/serialutil.py", line 282, in __init__
    self.open()
  File "/usr/local/lib/python2.7/site-packages/serial/serialposix.py", line 289, in open
    self.fd = os.open(self.portstr, os.O_RDWR|os.O_NOCTTY|os.O_NONBLOCK)
OSError: [Errno 16] Resource busy: '/dev/cu.usbmodem1451'
```

图 5-2　Python 无法使用设备的提醒

在开始之前，我们可以先为 Arduino 写一个串口读取程序。

```
void setup() {
  Serial.begin(9600);
}

void loop() {
  if (Serial.available()) {
    char inChar = (char)Serial.read();
    Serial.write(inChar);
  }
}
```

在 setup()函数中我们初始化了串口，其波特率是 9600。接着在循环函数中，如果串口可用，那么 Serial.available()的值就会等于 true，就进入了读取函数中。由于读取的值是 Byte，我们需要将其转换为字符。最后，在我们的串口中输出字符。在 Arduino IDE 中，我们可以打开串口监视器来查看 Serial.write 的值。

打开 Arduino IDE 的串口监视器，然后进入 Python 的交互命令模式，然后输入下面的代码：

```
import serial
ser=serial.Serial("/dev/cu.usbmodem1451",9600)
ser.write('hello')
```

首先，我们引入了 Python 的 serial 模块。然后，我们初始化了串口，其波特率是 9600，串口号是“/dev/cu.usbmodem1451”，在 Linux 系统上这个值可能是“/dev/ttyACM0”，在 Windows 上应该是“COM3”这一类的串口号。如果这个过程没有出错的话，我们会得到类似如图 5-3 所示的结果。

```
phodal:~ root# python
Python 2.7.10 (default, Jul 13 2015, 12:05:58)
[GCC 4.2.1 Compatible Apple LLVM 6.1.0 (clang-602.0.53)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import serial
>>> ser=serial.Serial("/dev/cu.usbmodem1451",9600)
>>> ser.write('hello')
5
>>>
```

图 5-3　Python 使用串口输入

在我们输入成功后，在 Arduino IDE 的串口监视器上会得到如图 5-4 所示的结果。

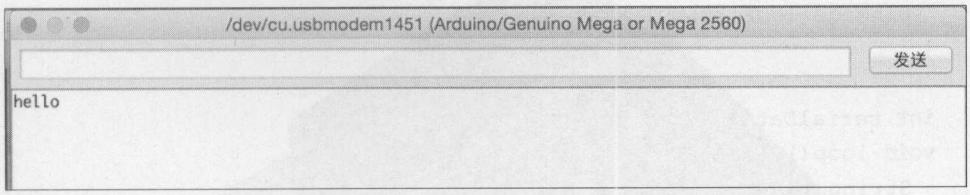


图 5-4 Arduino 串口输出 Hello

在完成上面的步骤后，我们就可以在计算机或者 Raspberry Pi 上进行操作了。建议读者运行在上一章中完成的服务器，这样你就可以真实地操作设备了。当我们将代码部署到服务器上的时候，就可以直接使用了。

```
import json
import urllib2
import serial
import time

url = "http://localhost:3000/api/1"
ser = serial.Serial("/dev/cu.usbmodem1451", 9600)

while 1:
    date = urllib2.urlopen(url)
    status = json.load(date)[0]['led']
    print status
    if status:
        ser.write('1')
    else:
        ser.write('0')
    time.sleep(1)
```

相信上面的代码除了 time，你基本上已经知道怎么用了，time 函数在这里的作用就是等待。等待一秒后，才继续发 HTTP 请求。

我们的 Arduino 做的代码便是接收上面的代码，解析其中的数据。如果接收到的数据等于 1，那么就点亮 LED。如果接收到的数据是 0，那么就置为低电平。

```
int ledPort=13;
```

```
void setup() {
  Serial.begin(9600);
  pinMode(ledPort,OUTPUT);
}

int serialData;
void loop() {
  String inString = "";
  while (Serial.available()> 0)
  {
    int inChar = Serial.read();
    if (isDigit(inChar)) {
      inString += (char)inChar;
    }
    serialData=inString.toInt();
    Serial.print(serialData);
  }
  if(serialData==1){
    digitalWrite(ledPort,HIGH);
  }else{
    digitalWrite(ledPort,LOW);
  }
}
```

在完成向 Arduino 烧录代码后，我们就可以在网页上操作 LED，然后观查开发板上 LED 的状态。如果你用的是其他单片机，也可以尝试连接到你的电脑，它们的原理是相同的。

我们也可以考虑使用其他模块与 Arduino 一起配合使用，如 ESP8266 模块，但是 ESP8266 模块在胜任协调层上存在一定难度。

5.1.3 Arduino 与网络模块

除了上面的方法，我们还可以尝试直接用 Arduino 使用网络模块连接到服务器。在 Arduino IDE 内置的 Ethernet 库就是专门针对网络功能而开发的。

如图 5-5 所示是 Arduino 的网络模块 W5100。

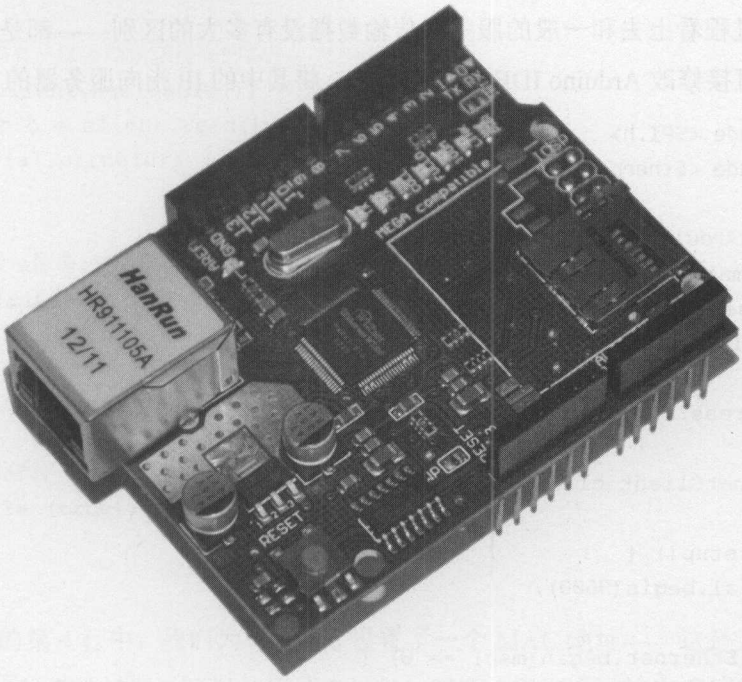


图 5-5　Arduino 网络模块

它是一款集 TCP/IP 协议、MAC 和 PHY 于一体的网络芯片，可以支持直接总线接口、间接总线接口和 SPI 总线。并且可以直接使用官方的 Ethernet 库。如图 5-6 所示是程序连接到我们的服务器并获取数据的截图。

```
/dev/cu.usbmodem1451 (Arduino/Genuino Mega c

connecting...
connected
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 59
ETag: W/"3b-h55mQ99mLOalZWSNb6wJ8A"
Date: Thu, 05 Nov 2015 14:50:41 GMT
Connection: close

[{"_id":"562f2a732c497994b03f83fc","user":"1","led":false}]
disconnecting.
```

图 5-6　Arduino 连接 Web 服务器时的日志

这个过程看上去和一般的服务器传输数据没有多大的区别——都是字符串处理。我们可以直接修改 Arduino IDE 的示例代码，将其中的 IP 指向服务器的 IP：

```
#include <SPI.h>
#include <Ethernet.h>

//设置 Arduino 的 Mac 地址
byte mac[] = { 0xDE, 0xAD, 0xBE, 0xEF, 0xFE, 0xED };
char server[] = "192.168.3.12"; //即服务器的 IP，由于是连接到本地的计算机上，
所以是 192.168.3.12

IPAddress ip(192,168,3,200); //动态分配 IP 失败时使用的静态 IP

EthernetClient client;

void setup() {
  Serial.begin(9600);

  if (Ethernet.begin(mac) == 0) {
    Serial.println("Failed to configure Ethernet using DHCP");
    Ethernet.begin(mac, ip);
  }

  delay(1000);
  Serial.println("connecting...");

  if (client.connect(server, 3000)) {
    Serial.println("connected");
    //创建 HTTP 请求
    client.println("GET /api/1 HTTP/1.1");
    client.println("Host: 192.168.3.12");
    client.println("Connection: close");
    client.println();
  }
  else {
    Serial.println("connection failed");
  }
}

void loop()
```

```
{
  //如果有从服务器请求过来的数据，直接读取并显示
  if (client.available()) {
    char c = client.read();
    Serial.print(c);
  }

  //如果从服务器断开，停止客户端
  if (!client.connected()) {
    Serial.println();
    Serial.println("disconnecting.");
    client.stop();

    //死循环
    while (true);
  }
}
```

在代码的第4行中，我们为Arduino设置了一个MAC地址——这是为了当芯片没有默认的MAC地址时，可以使用这个地址。随后，我们通过输出下面的字符串来和服务器进行通信，即上面代码中的第25~28行。

```
"GET /api/1 HTTP/1.1"
"Host: 192.168.3.12"
"Connection: close"
```

最后，当我们连接上服务器的时候，将输出这些数据。

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 58
ETag: W/"3a-dVHcf3cRbNcPb87P4+KBmg"
Date: Wed, 11 Nov 2015 15:02:30 GMT
Connection: close

[{"_id":"56418b06bcfa06f7ab9fa022","user":"1","led":true}]
```

但是直接使用 Arduino 连接到网络并解析数据不是一件容易的事。由于在Arduino上处理这个请求本身会花费相当长的计算时间，使得整个系统的实时性特别

差，在这里不推荐使用 Arduino 直接连接网络。我们可以选用 Raspberry Pi 来处理 HTTP 协议，或者也可以使用 ESP8266。

5.1.4 NodeMCU

NodeMCU 是一款基于 ESP8266 的开源快速硬件原型平台——包含固件和开发板。ESP8266 是一块由乐鑫 espressif 设计的 WiFi 芯片，在一些情景中，我们可以将之视为物联网解决方案。这是一块造价便宜而性能强大的芯片。

现在让我们来看看这个开发板是如何连接到网络的。下面的代码是 NodeMCU 用于获取同样请求的代码：

```
-- A simple http client
conn=net.createConnection(net.TCP, false)
conn:on("receive", function(conn, pl) print(pl) end)
conn:connect(3000,"192.168.3.12")
conn:send("GET / HTTP/1.1\r\nHost: 192.168.3.12\r\n"
    .."Connection: keep-alive\r\nAccept: */*\r\n\r\n")
```

可以看到上面的代码言简意赅——先创建了一个 TCP 连接，然后在第 2 行开始监听是否接收到消息，并输出这些消息。接着连接 192.168.3.12 的 3000 端口，最后发出相应的 GET 请求。

与 C 语言相比，上面的代码更具表现力、简洁、清晰。如果只是一个单一的功能，例如读取上传传感器的值、控制某一个单一的设备，这个开发板是一个很不错的选择。

Arduino ESP8266

Arduino core for ESP8266 是在 Github 上的一个开源项目：https://github.com/esp8266/ Arduino，旨在为 ESP8266 芯片提供 Arduino 环境。直接使用 ESP8266 芯片作为控制器，而不仅仅只是使用 ESP8266 的网络连接功能。与 Arduino 相比，它提供了更多的功能，如 WiFi，并且可以作为 HTTP、mDNS、SSDP、DNS 服务器，还可以实现 OTA 更新。

在使用之前，我们需要最新的 Arduino IDE（1.6.4 版本以上）。启动 Arduino IDE，

并打开 Preferences，在 Additional Board Manager URLs 中输入 http://arduino.esp8266.com/stable/package_esp8266com_index.json。确定后再打开工具→板→“Boards Manager”，找到 esp8266 那一行单击 INSTALL，如图 5-7 所示。

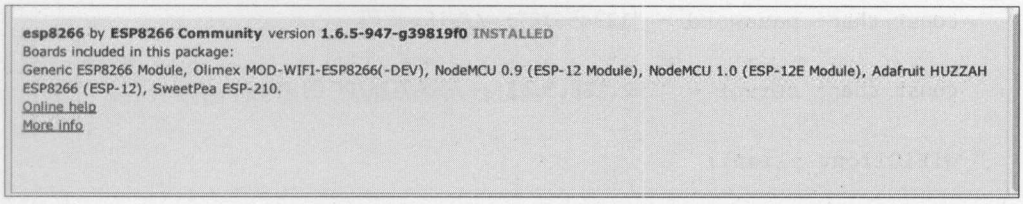


图 5-7　Arduino 开发板管理器

如果中途遇到网络错误，请尝试重试几次或者使用 VPN。其连接到服务器的代码与 Arduino 的代码并无太大的区别。唯一不同的可能是这里的连接方式是 WiFi，需要先配置好 WiFi 的 SSID（即你所使用的路由器的 WiFi 名称）和密码。

安装完后，重启 Arduino 就可以生效。如图 5-8 所示，我们可以在工具菜单中选择相应的 ESP8266 模块，如果没有 ESP8266 模块，可以选择 **Generic ESP8266 Module**。

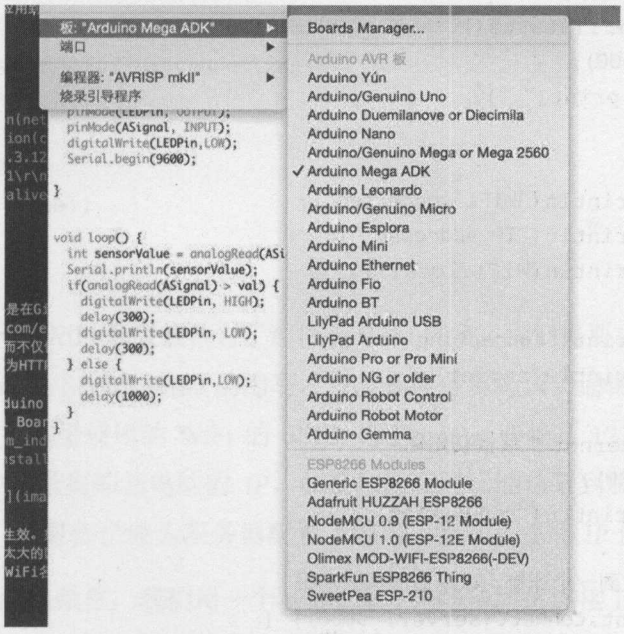


图 5-8　成功安装后出现相应的 ESP8266 模块

现在，我们可以编写一个程序来获取 API 的数据，并将这个过程打印出来。

```
#include <ESP8266WiFi.h>

const char* ssid     = "Phodal"; //WiFi 名称
const char* password = "12345678"; //WiFi 密码

const char* server = "192.168.3.12"; //服务器/PC 的 IP

WiFiClient client;

void setup() {
  Serial.begin(9600);
  delay(10);

  Serial.println();
  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());

  Serial.print("connecting to ");
  Serial.println(server);

  //等待 Ethernet 扩展板初始化
  delay(1000);
  Serial.println("connecting...");

  //如果获取到一个连接，以串口的形式汇报
  if (client.connect(server, 3000)) {
    Serial.println("connected");
```

```
    //创建一个 HTTP 请求
    client.println("GET /api/1 HTTP/1.1");
    client.println("Host: 192.168.3.12");
    client.println("Connection: close");
    client.println();
  }
  else {
    Serial.println("connection failed");
  }

}

void loop() {
  //如果有从服务器返回数据，读取并显示它们
  if (client.available()) {
    char c = client.read();
    Serial.print(c);
  }

  //如果服务器断开，停止客户端连接
  if (!client.connected()) {
    Serial.println();
    Serial.println("disconnecting.");
    client.stop();

    //无限循环
    while (true);
  }
}
```

由于这是一个 WiFi 开发板，为了连接上 WiFi 网络，我们需要先设置 SSID 和密码。SSID 是 Service Set Identifier 的缩写，意思是服务集标识，通俗来说就是要连接的 WiFi 的名称。这里使用的 WiFi 的 SSID 是 Phodal，密码是 12345678。server = "192.168.3.12"中的数据即我电脑的 IP，如果你的应用已经部署到服务器上，便是托管服务器的 IP。如果你已经为服务器配置了域名，那么可以将 IP 修改为你的域名。

接着在 setup 函数里，我们用一个 while 循环来等待网络连接上，即如下所示的代码：

```
while (WiFi.status() != WL_CONNECTED) {
  delay(500);
  Serial.print(".");
}
```

在连接上 WiFi 后，我们将输出本机的 IP。

随后，我们声明了一个 client 的 WiFiClient 对象，我们将用这个对象去连接服务器。并向服务器发送对应的请求，下面的代码便是 GET /api/1 的请求的内容——都是以字符串的形式发出去的。这里的代码和上面的 Arduino 网络模块的示例是一样的。

```
client.println("GET /api/1 HTTP/1.1");
client.println("Host: 192.168.3.12");
client.println("Connection: close");
client.println();
```

当服务器做出响应的时候，我们将输出响应的内容，结果如图 5-9 所示。

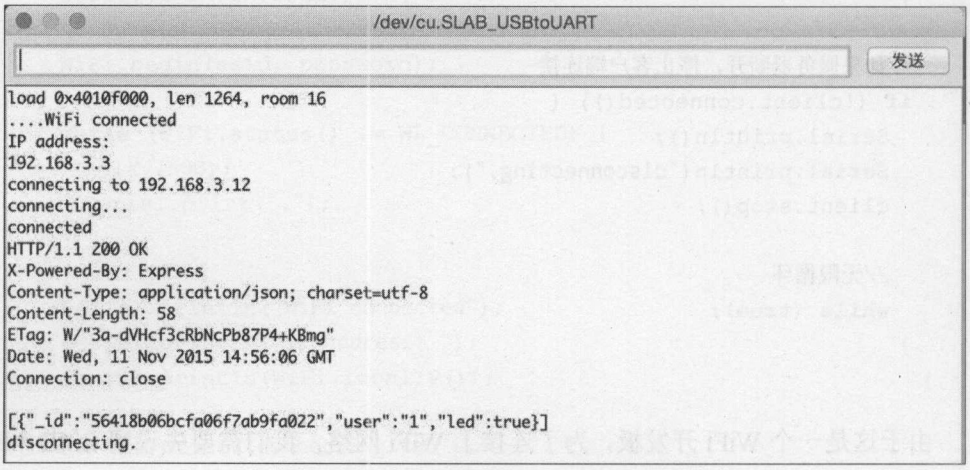


图 5-9　ESP8266 Web 连接时的日志

除了这两类典型的结构外，不得不提到的一种方式是——手机通过蓝牙与硬件相连接。这也是一种很理想的方式，而且开发过来也相当快捷。我们会在稍后的章节中，做一个简单的 Demo 来向读者展示其中的美妙之处。

5.2 连接执行器

在上节中，我们的控制器已经可以连接到服务器，并且在一些例子中甚至可以直接控制 Led 了。但是这远远不够，要知道我们必须用它们来控制真正的电器。

在实施联网的过程中会有一个很严重的问题一直困扰着我们——设备多样性。而这个问题一直困扰着很多人，从早期的大型机的开发，到现在的各式各样的移动设备——不仅操作系统不一样，屏幕大小也不一样。同理，对于现实世界的设备来说也是如此，而在物联网领域这个问题更加严重。

如果我们只想去控制空调，那么首先我们只需要控制好我们使用的那个空调就好了。然后，我们发现了这个空调的厂商的接口几乎都是一样的，除了少数有点不同。接着，我们发现了需要处理更多的不同厂商的空调。最后，我们发现要处理的不仅仅有空调，还要有冰箱、电暖、窗帘等。

这一切到后面会成为困扰我们的问题，尽管可能会有新的措施出台来保证它们之间要使用相同的协议等，但是要更换所有旧的设备不是一朝一夕就能完成的事。

我们控制执行器的方式有两种：

（1）直接控制。这是一种很理想的方式，但是这取决于设备的使用方式。通常我们很难控制现在的设备，因为这些设备面向的接口都是人。这种控制类型最适合的要数电灯了，它只有开和关。

（2）间接控制。多数情况下，我们采用这种方案。如我们使用遥控器来控制电视、空调，遥控器本身也是间接控制的，这也意味着我们可以编写程序来控制这些设备。

现在让我们从直接控制开始。

5.2.1 直接控制示例

出于安全考虑，在这里我们先用其来控制一些电流较小的电器，如一个 Arduino 开发板的电源。这时我们就需要用继电器来做这些工作。

继电器是一种电子控制器件，它具有控制系统和被控制系统。常见于自动控制电路中，是一种用小电流去控制较大电流的一种“自动开关”，如图 5-10 所示。按照其工作原理或特征，人们又将其分为电磁继电器、固体继电器、机械继电器、热敏干簧继电器等。如小米智能插座在内部采用 250V 10A 机械继电器对电源输出插口进行控制。

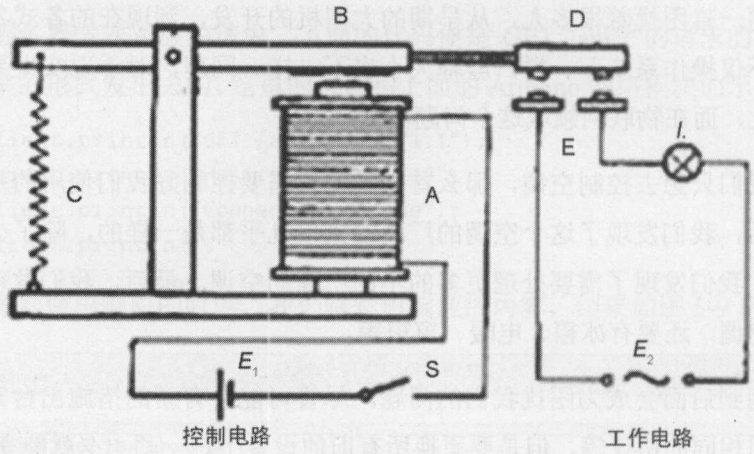


图 5-10　继电器工作原理图

当电磁铁通电时，衔铁将被吸下来使 D 和 E 接触，工作电路闭合。电磁铁断电时失去磁性，弹簧把衔铁拉起来，切断工作电路。在控制一些生活电器的时候，我们都需要使用继电器来辅助我们完成这些功能。在本书的原计划中打算以控制电灯泡作为示例，随后考虑到其可能会带来一些安全问题，便改用 Arduino 来控制另外一个 Arduino 的供电。在购买继电器时，请注意查看上面标注的工作电压范围。

从开关的种类又可以将其分为单刀单掷（SPST）、单刀双掷（SPDT）、双刀单掷（DPST）、双刀双掷（DPDT）等。这里我们使用的是单刀双掷（SPDT）：它使用一个线圈来操作有三个触点的开关。图 5-11 是我们将用到的继电器的实物图。

其输出部分有三个接线柱，分别是 NC、NO、COM。COM 全称则是 Common，表示共接点。NC 全称是 Normal Closed，即常闭接点，当线圈通电后才成为开路（断路）。NO 全称是 Normal Open，即常开接点，只有线圈通电后才与共接点 COM 接通。

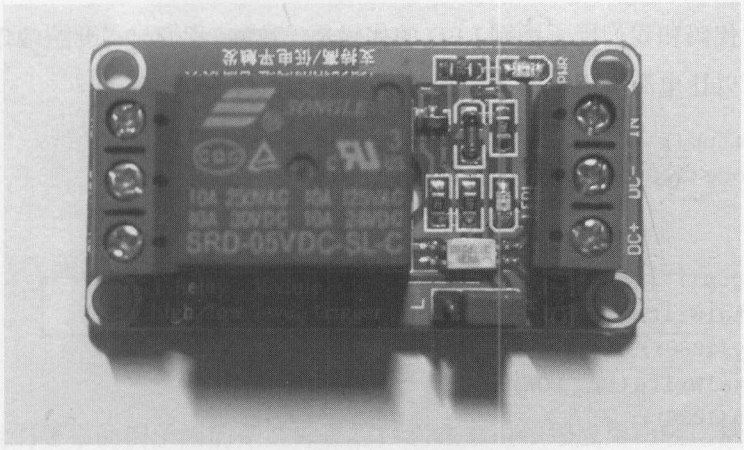


图 5-11　继电器实物图

其输入部分为 DC+、DC-和 IN，即电源正极、电源负极、输入口——也是 Arduino 的输出口。

在输入部分，连接 Arduino 的 5V 到 DC+、GND 到 DC-、11 引脚到 IN。在输出部分，连接 COM 口到电源的正极、NO 连接到用电器的正极。最后，电路连接图如图 5-12 所示。

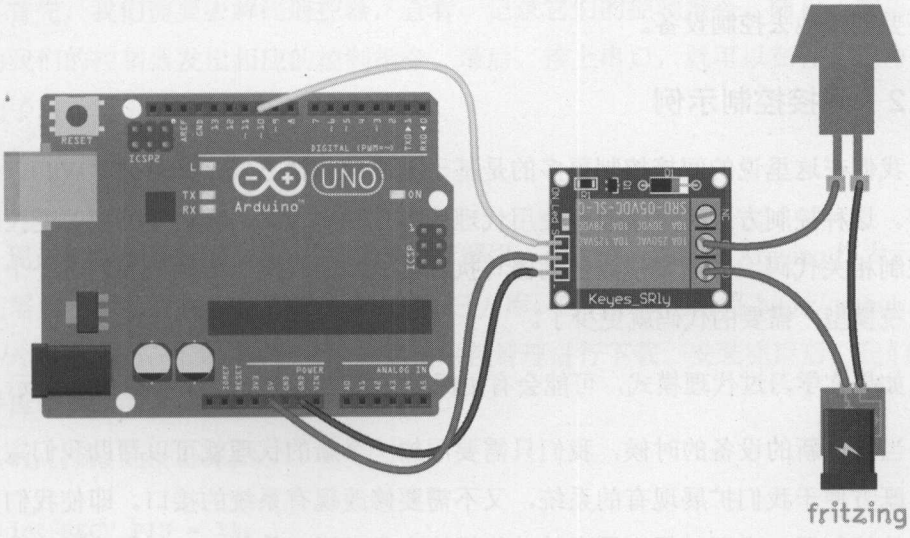


图 5-12　电路连接图

我们的代码和我们用于控制 LED 的代码是一致的，我们只需要给端口 11 一个高电平，就可以让电源与用电器接触。

```
void setup() {
  pinMode(11, OUTPUT);
}

void loop() {
  digitalWrite(11, HIGH);
  delay(1000);
  digitalWrite(11, LOW);
  delay(1000);
}
```

这和 Arduino IDE 最开始的闪烁（Blink）示例几乎是一致的，除了端口不一样。如果我们的继电器支持，可以将电源改成 220V，用电器变成灯泡。如果你要进行这种超过安全用电（36V）的实验，请在有经验人员的陪同下进去，或者进行相应的安全措施。

当我们直接控制设备的时候，我们需要考虑设备与控制器的连接方式、交互方式等。很多时候由于某些原因，如距离，我们无法直接地去控制设备，这时候我们就需要间接地去控制设备。

5.2.2 间接控制示例

我们在这里说的间接控制更多的是基于无线的传播方式，如蓝牙、WiFi、红外线等。这种控制方式有点类似于使用代理的方式控制设备，我们不需要改变控制器的控制相关代码，只需要修改控制器与执行器之间的代码。而如果我们有一个更好的系统模型，需要的代码就更少了。

如果你学习过代理模式，可能会有更深刻的感觉。这个过程如图 5-13 所示。

当出现新的设备的时候，我们只需要添加一个新的代理就可以帮助我们完成工作。既方便于我们扩展现有的系统，又不需要修改现有系统的接口。即使我们更换了新的控制器，也可以保证原有的功能模块也是可以工作的——如果我们新设计的

控制电路可以支持现有的通信模式。

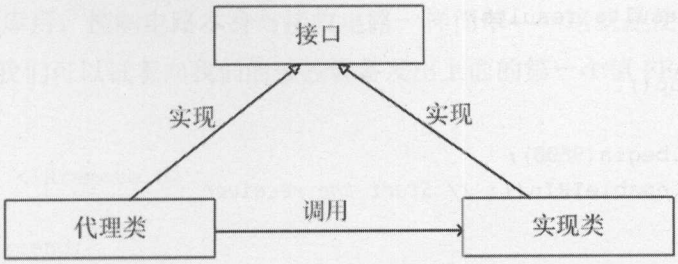


图 5-13 代理模式

这里我们要举的例子用的是红外遥控，它在家电领域应用比较广泛。它是一种无线、非接触控制技术，并且简单易用且实现成本较低。由于其直射特性，红外通信技术不适合传输障碍较多的地方。如在常见的电视、空调遥控器中，如果你的遥控器和你的设备之间有障碍物，那么你就无法直接控制设备了。

遥控器一般可以由红外接收及发射电路、控制板、键盘及状态指示电路组成。当我们可以成功地用控制器向空调、电视机等发出控制信号的时候，我们就可以直接在电脑上、手机上直接控制这些电器了。

首先，我们需要去解码遥控器，查看、记录它们的控制指令。随后，就可以直接由我们的控制器发出相应的控制指令。最后，接上串口，就可以在任何地方控制它们了。

1. 解码

现在我们对需要进行控制的遥控器进行解码。在这里，仍然以 Arduino 作为示例，我们需要一个名为 Arduino IRremote 的第三方库。它的下载地址是 https://github.com/z3t0/Arduino-IRremote，你也可以直接使用包管理进行下载。安装完库后，我们可以打开库中自带的示例程序：

```
#include <IRremote.h>

int RECV_PIN = 11;
```

```
IRrecv irrecv(RECV_PIN);

decode_results results;

void setup()
{
  Serial.begin(9600);
  irrecv.enableIRIn(); // Start the receiver
}

void loop() {
  if (irrecv.decode(&results)) {
    Serial.println(results.value, HEX);
    irrecv.resume(); // Receive the next value
  }
  delay(100);
}
```

程序引入了 IRemote 这个库，并且设置了接收的引脚是 11 端口。然后在 setup 函数里，初始化了红外接收。最后在 loop 函数里，处理输入的信号并进行解码、串口输出。

接着，可以按下我们的遥控器，并记录对应的按键值，如图 5-14 所示。

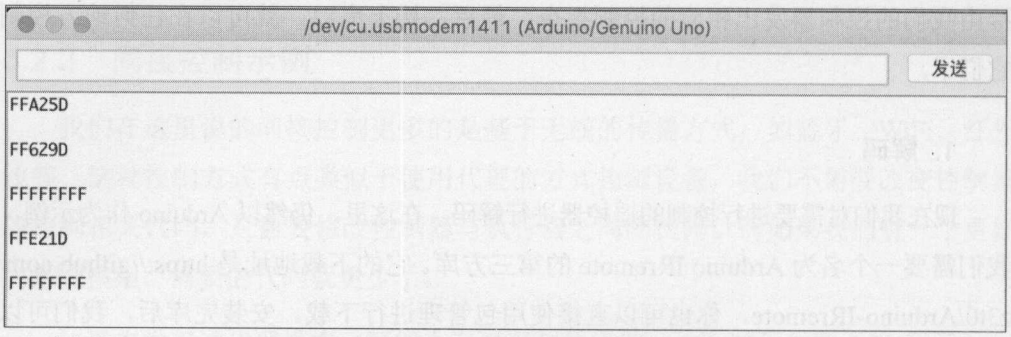


图 5-14　Arduino IRemote 接收编码

这里的 FFA25D 对应的是第一个按键，FF629D 对应了第二个按键，等等。这里出现 FFFFFFFF 是因为我长按了按键。随后，我们就可以以同样的指令去控制设备了。

2. 控制

在引入了库后，控制电路本身与接收电路一样简单——这就是使用了库带来的优点。现在，我们可以试着向我们的被控设备发出上面的第一个值 FFA25D。代码如下所示：

```
#include <IRremote.h>

IRsend irsend;

void setup()
{
}

void loop() {
   for (int i = 0; i < 3; i++) {
      irsend.sendNEC(0xFFA25D,32);
      delay(40);
   }
   delay(5000);
}
```

我们需要做的就是初始化 irsend，然后就可以发出相应控制器的指令。这里的 sendNEC 是因为使用的通信协议是 NEC 公司的——不同的厂商都有自己专属的红外通信协议，常见的有 NEC、SONY、Philips。需要根据所需控制器的厂商来选择相应的函数。如果你所要控制的设备并没有在上面，那么你可能需要通过一些设备，如示波器，来完成相应的工作。

5.2.3 示例代码

现在，让我们来结合上面的代码来搭建执行端。当我们从串口输入不同的数字的时候，我们来执行不同的操作：

- 当输入 1 时，给予继电器常开口高电平。
- 当输入为 2 时，发出红外指令。
- 当输入为 3 时，给予继电器常闭口高电平。

这部分代码如下所示：

```
#include <IRremote.h>

int port = 12;
IRsend irsend;
int startArduino = 1;
int irRemote = 2;
int stopArduino = 3;

void setup() {
  Serial.begin(9600);
  pinMode(port, OUTPUT);
}

int serialData;
void loop() {
  String inString = "";
  while (Serial.available()> 0)
  {
    int inChar = Serial.read();
    if (isDigit(inChar)) {
      inString += (char)inChar;
    }
    serialData=inString.toInt();
    Serial.print(serialData);
  }
  if(serialData == startArduino){
    digitalWrite(port, HIGH);
  } else if( serialData == irRemote){
    irsend.sendNEC(0xFFA25D,32);
    delay(1000);
  } else if (serialData == stopArduino) {
    digitalWrite(port, LOW);
  }
}
```

在这里我们会用 isDigit 方法来判断输入的字符串是否是数字，如果不是数字，将把这个值转换为数字，然后再判断。其实物连接图如图 5-15 所示。

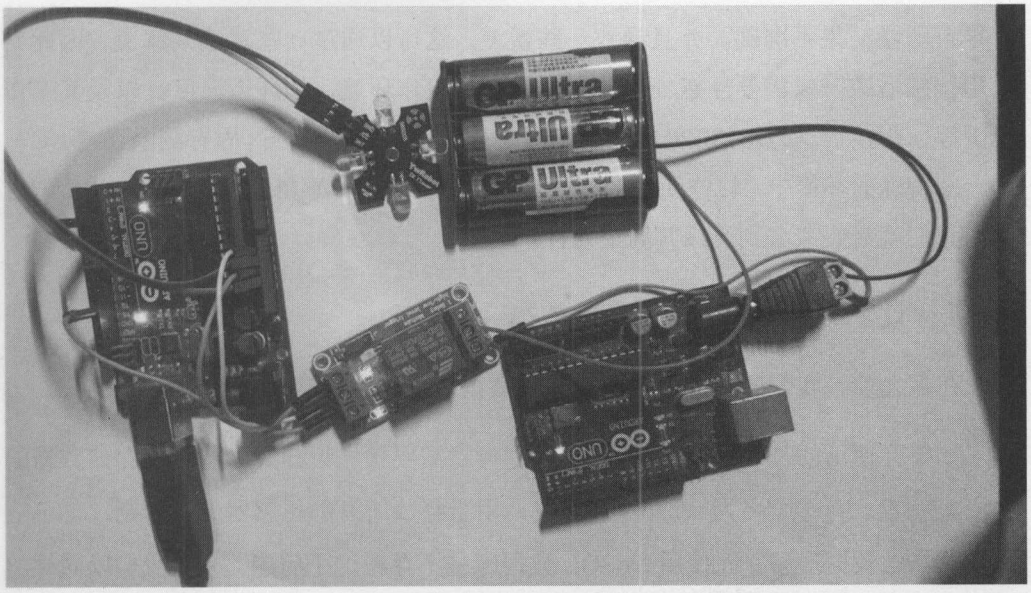


图 5-15　控制器示例

有了这些基础知识，我们就能控制大部分执行器了。在控制之前，我们需要一些逻辑基础——传感器。

5.3　连接传感器

传感器测量的是真实世界的物理量，在测量完后，我们需要将这些数值上传到服务器。这些数据提供给其他控制器或者协调层来处理、分析数据，并发出相应的控制指令。如 Oculus[1] 公司在面向开发人员的虚拟现实设备 Oculus Rift 中采用了 MPU6000 六轴陀螺仪传感器——融合 3 轴陀螺仪、3 轴加速器，传感器检测用户头部在空间中的运动情况，获取其用于表示空间旋转状态的四元数与欧拉角，再分析这些数值上传到计算机来控制屏幕中物体的状态。同理，在手机上人们通过倾斜屏幕来玩游戏也是这个原理。

需要注意的是，由于传感器属于电子元件，可能易受外界的一些干扰。这时候

1 一款为电子游戏设计的头戴式显示器。

会导致测量结果不准确，并且导致一些误差，这可以通过一些算法来解决。因此在选用传感器的时候需要注意一下传感器适合的工作环境，如环境温度，以及其工作范围等一些参数。如DS18B20的测温范围是从 −55℃～+125℃，固有测温误差1℃。在一些特殊的环境中，这些元件会相互影响，如当这个传感器需要工作在100℃的时候，我们就需要考虑到升级其周边元件。

5.3.1 让API支持上传传感器数据

在那之前，我们的服务器需要能处理数据，这就意味着要先修改一下API。

如果你使用的是SQL数据库，那么你可能需要预先设计好这些用于存储数据的字段。或者，有一个名为sensors的字段，用于存放所有的传感器值。否则，你可能需要经常做数据迁移。而使用NoSQL数据库就不存在这个问题。只要我们以同样的方式存储数据，并以同样的方式读取数据即可。

因此我们需要更新UpdateData方法。原先，我们会从request的body中拿到led的值并判断：

```
var payload = {user: req.params.user_id};
var data = {user: req.params.user_id, led: false};
if (req.body.led === true) {
    data.led = true;
}
```

现在，我们假设客户端返回的数据是正确的，即不对其数据进行验证。客户端想要存储什么数据就存储什么数据，在读取的时候返回两样的数据。修改完后的代码如下：

```
function updateData(req, res) {
    var userId = req.params.user_id;
    var payload = {user: userId};

    var data = req.body;
    data.user = userId;
```

```
        db.find(payload, function (results) {
            if (results.length > 0) {
                db.update(data);
                res.send({db: "update"});
            } else {
                db.insert(data);
                res.send({db: "insert"});
            }
        });
    }
```

直接将 req.body 的值赋予 data，并将 userId 也保存了下来——为了以后查询和修改。现在，让我们来更新数据试试：

```
    curl -X PUT -d '{ "led": true, "temperature": 33 }' -H "Content-Type:
application/json" http://localhost:3000/api/14
    curl -X PUT -d '{ "led": true, "temperature": 33, "sun":13 }' -H
"Content-Type: application/json" http://localhost:3000/api/15
```

我们创建了两个新的数据，一个是 user 为 14，另一个是 user 为 15，打开 MongoDB 的客户端图形界面，会有如图 5-16 所示的内容。

Key	Value	Type
▼ (1) ObjectId("56402d546c...	{ 4 fields }	Object
_id	ObjectId("56402d546c1c7d1670...	ObjectId
led	true	Boolean
temperature	33	Int32
user	14	String
▼ (2) ObjectId("56402d586c...	{ 5 fields }	Object
_id	ObjectId("56402d586c1c7d1670...	ObjectId
led	true	Boolean
temperature	33	Int32
sun	13	Int32
user	15	String

图 5-16　MongoDB 返回不同 Key 的结果

第二个对象比第一个对象多了一个 sun 的键值。对于物联网应用来说，灵活的数据库结构更容易加速我们的开发过程，并且方便扩展。

在我们完成这部分的扩展之后，就可以接上我们的传感器并开始传输数据。

5.3.2 土壤湿度传感器

除了温度传感器，土壤湿度传感器也是一个很实用的传感器。不过对于那些并不喜欢养植物的人来说，可能并不实用。

Moisture Sensor 是一个简易水分传感器，利用电阻变化测量水分，土壤越干燥，输出电压越小；反之，越大。其工作电压为 3.3V~5V，输出电压 0V~3.6V（5V 供电），使用时只需将 Moisture Sensor 传感器探头插入土壤中，通过 AD 转换电压信号，即可检测土壤中的水分。

这里我们需要一个土壤湿度传感器，以及一个用于调节土壤湿度的阈值的电位器。这个传感器是利用两个大面积的接触面引脚，来探测土壤中水分的含量。当土壤中水的含量越高的时候，引脚间的导电率越高，其输出阻值就越小；当土壤中水分比较少时，其阻值越大。元件连接方式如图 5-17 所示。

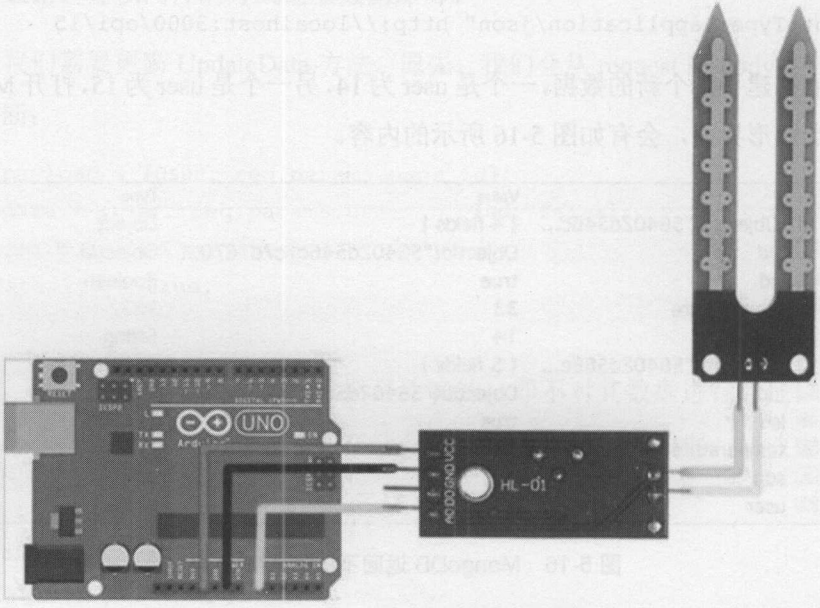


图 5-17 Arduino Moisture Sensor 连接方式

我们将 A0 引脚连接到 Arduino 上的 A0，GND 和 VCC 分别连接到对应的引脚上。下面是我们编写的用于获取这个传感器值的代码：

```
int DataPin = A0;
int LEDPin = 13;
int limitValue = 380;

void setup() {
  pinMode(LEDPin, OUTPUT);
  pinMode(DataPin, INPUT);
  digitalWrite(LEDPin,LOW);
  Serial.begin(9600);
}

void loop() {
  int sensorValue = analogRead(DataPin);
  Serial.println(sensorValue);
  if(analogRead(DataPin) > limitValue) {
    digitalWrite(LEDPin, HIGH);
    delay(500);
    digitalWrite(LEDPin, LOW);
    delay(500);
  } else {
    digitalWrite(LEDPin,LOW);
    delay(1000);
  }
}
```

我们定义的数据引脚是A0，用LED来显示当前的状态——当测量到的值低于设定的值的时候，将闪烁LED，并且将在控制台输出这个值。在这里，可以用Raspberry Pi读出这个值，并将这个值存储到数据库中。

5.3.3 温度传感器

温度传感器是一个很实用并且常见的传感器。我们将使用温度传感器作为一个简单的例子，来上传温度传感器的数据。

这里使用的温度传感器是DS18B20，它是由美国DALLAS公司生产的单总线数字温度传感器。这是世界上第一片支持“一线总线”接口的温度传感器，它仅需一条线（和地线）就可以将数据送入MCU。因为它只能按照DALLAS规定的一种时

序才能正确传出数据，故在这里我们使用官方的库来进行演示。

1. 使用第三方库

与土壤湿度传感器稍有不同的是，温度传感器需要依赖于第三方开发的库来开发。这并不意味着我们一定要使用第三库，只是因为使用第三方库会加速我们开发的过程。在这个库中可能会发现一些二进制数据，等待几毫秒，接收数据、排序并进行一些计算，最后返回数据。并且通常这些库会抽象出一些简单并且容易使用的函数来方便我们开发。

在学习的时候使用库是一个很明智的做法，它可以为我们带来很多好处：

- 开源而且免费。这里的开源更多针对的是 Arduino 相关的开发板。由于 Arduino 的开源浪潮，使得这个平台上的绝大多数库都以开源的形式发布出来。如果这个库在实现上有一些缺陷，那么也可以修改其中的代码来实现我们的功能。
- 程序变得简单。这些库会使我们写的代码更简洁——我们只需要调用函数即可。
- 开发更快。例如在使用这个温度传感器的时候，我们想要的是获取真实世界的温度值。只要这个值和我们在温度计上看到的一致，那么就不会关心它的代码是不是写得很烂。
- 节省时间。
- 稳定。这里的稳定只是相对的，有些第三库功能和代码实现得比较好——它经过一些用户的反馈和改进。在我们自己实现的过程中，可能写不出这么好的代码。

相比之下，当然也有一些缺点：

- 不稳定。由于这里的库大部分是一些爱好者在业余的时间做出来的，一些库可能没有经历过稳定的测试。
- 代码质量不高。

是的，我只想到了这个缺点。因为大部分 Arduino 相关的库都是开源的，这意味着即使这个库的稳定性、实用性达不到我们想要的程度，也可以在这个库的基础上进行修改。而稳定性不好可能不仅仅是作者一个人的问题，笔者到现在一直在维护几个开源的软件，并且这些软件也有些用户。这些软件托管在 Github 上，按照习惯，

如果你使用了这些软件，并且遇到问题可以以 issue 的形式向作者或社区求助。而一般情况下，我都很少收到这种 issue——作为软件的开发者，我很难意识到这里面有什么潜在的问题。

在 Arduino 相关的开发社区中，Github（https://github.com/）就是一个很不错的网站，在上面你可以搜索到一些很实用的第三方库，如 Arduino 官方也将它们的 IDE 及硬件相关的代码托管在这上面。并且上面不仅仅会有硬件相关的库，也可以搜索到一些物联网、软件等相关的开源软件。人们出于不同的目的——爱好、分享、学习、备份——将他们的代码上传到 Github 上。

2. 导入第三方库

在这里，我们会用到两个第三方库：

（1）OneWire。顾名思义，即 1-Wire 总线系统使用的库。这里的 OneWire 则是由 Paul Stoffregen 开发的，专用于 Arduino 开发板的库。下载地址：http://www.pjrc.com/teensy/ arduino_libraries/OneWire.zip。

（2）Arduino Temperature Control Library。从这个库的名字中，你已经知道了这个库是用于 Arduino 温度控制的库，它面向的设备是 DS18B20、DS1822、DS1820 等。下载地址：https://github.com/milesburton/Arduino-Temperature-Control-Library。

现在，需要将上述的两个库导入 Arduino IDE 中。在最新的 Arduino IDE 中，已经集成库管理的功能，这就意味着可以打开项目→Include library→Library Manager 来打开如图 5-18 所示的对话框，在输入框中输入 onewire，便会出现 OneWire 相关的库。

选中相应的库，在右边单击安装按钮就可以安装相应的库。

除此，我们也可以在上面的网址中下载相应的库，并用其中的 Add .ZIP Library 来添加库，如图 5-19 所示。

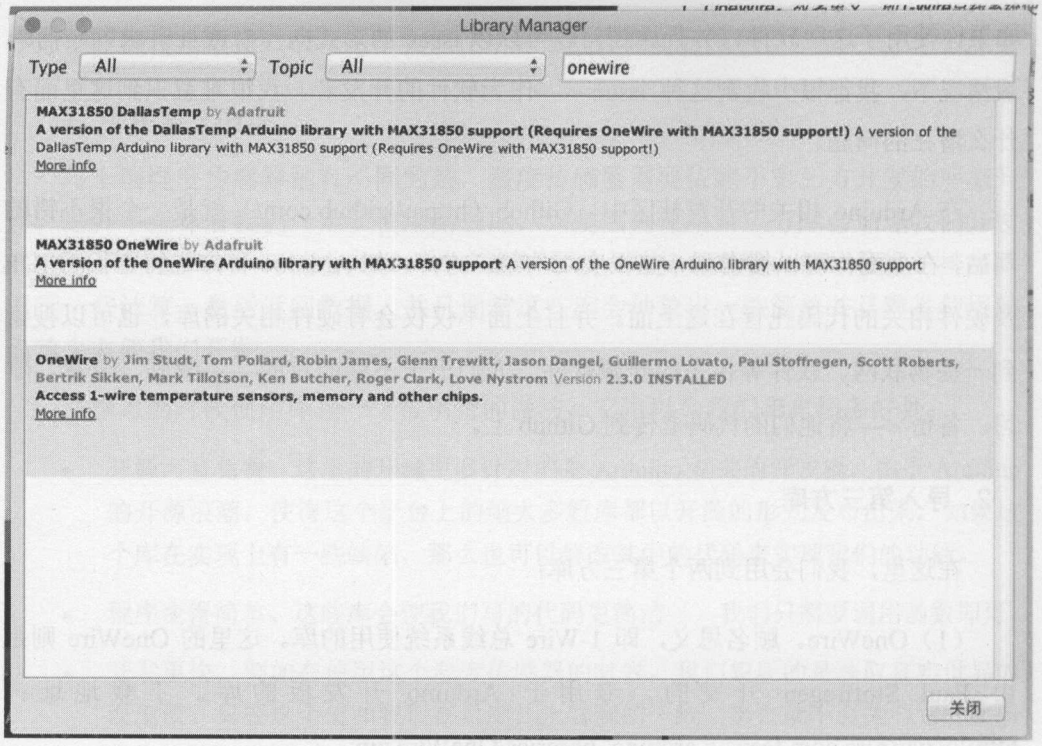


图 5-18　Arduino 库管理器截图

图 5-19　Arduino 添加.ZIP 库

现在，让我们来读取这个传感器的值！

3. 读取温度传感器

我们所需要的电路连接如图 5-20 所示。

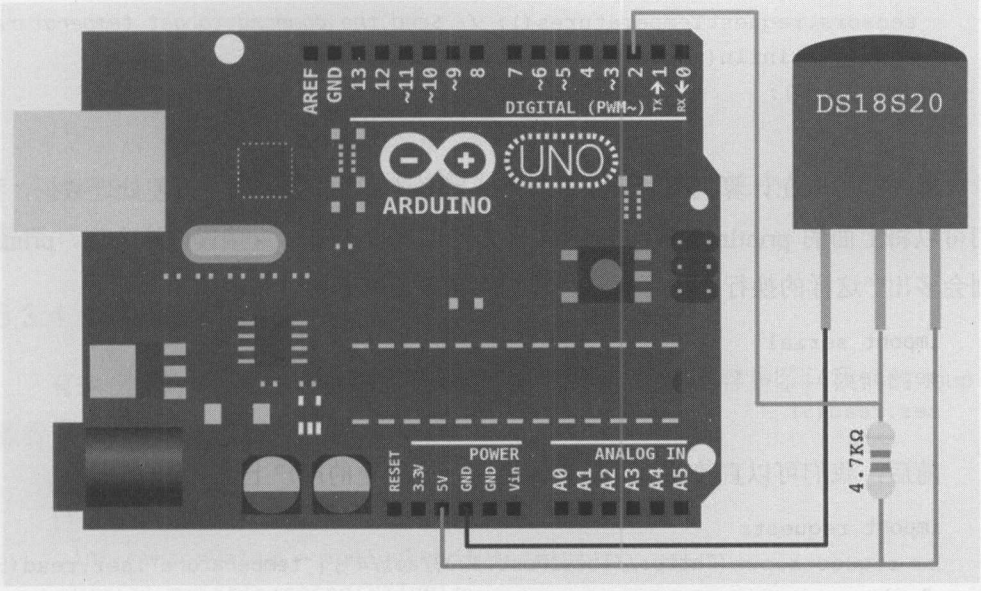


图 5-20　Arduino 与温度传感器连接电路图

我们需要在 1-Wire 数据线和 5V 电源之间添加一个大概 kΩ 的上拉电阻。并连接数据线到 Arduino 的第 2 个引脚。

```
#include <OneWire.h>
#include <DallasTemperature.h>

#define ONE_WIRE_BUS 2

OneWire oneWire(ONE_WIRE_BUS);

DallasTemperature sensors(&oneWire);


void setup(void)
{
  Serial.begin(9600);
  sensors.begin();
}

void loop(void)
{
```

```
    sensors.requestTemperatures(); // Send the command to get temperatures
    Serial.println(sensors.getTempCByIndex(0));
    delay(500);
  }
```

现在，可以在计算机或者 Raspberry Pi 上读取上面的值。为了方便处理数据，我们可以将上面的 println 改成 print——因为可以从 PySerial 直接读取这个值，println 则会多出""这样的换行。如下所示，我们将读到传感器传过来的值。

```
import serial
ser=serial.Serial("/dev/tty.usbmodem1411",9600)
ser.read(5)
```

随后，我们可以直接将这个值更新到某一个特定的用户上。

```
import requests
r = requests.put("http://localhost:3000/api/4",{"temperature":ser.read(5),
"user":4})
r.text
```

这时才发现，原来我们的数据库 update 方法有问题（笔者也是这时才发现的）——我们的代码只会去更新 LED 的状态。

```
collection.update({user: payload.user}, {$set: {led: payload.led}},
function (err, result) {
    callback();
});
```

现在，我们需要将$set 的值换成 payload，完整的 update 代码如下所示：

```
MongoPersistence.prototype.update = function (payload) {
    'use strict';
    MongoClient.connect(url, function (err, db) {
        var updateDocument = function (db, callback) {
            var collection = db.collection("documents");
            collection.update({user: payload.user}, {$set: payload},
function (err, result) {
                callback();
            });
        };
        updateDocument(db, function () {
```

```
            db.close();
        });
    });
};
```

接下来，我们可以不断地从串口获取传感器的值，并将这个值存储到我们的数据库中。

5.3.4 数据合并

有执行器的例子后，要整合传感器的例子已经变得相当容易了，我们的 loop 函数看上去可能就是这样的：

```
void loop(void)
{
  int sensorValue = analogRead(DataPin);
  Serial.println(sensorValue);

  sensors.requestTemperatures(); //发送读取温度的指令
  Serial.println(sensors.getTempCByIndex(0));

  delay(500);
}
```

你可能会意识到这其中会有一些问题——我们没有办法在协调装置上区分这些不同的值。于是，我们可能改成这样：

```
void loop(void)
{
  int sensorValue = analogRead(DataPin);
  sensors.requestTemperatures(); //发送读取温度的指令

  Serial.print(sensorValue);
  Serial.print("\t");
  Serial.println(sensors.getTempCByIndex(0));

  delay(500);
}
```

随后，通过空格来区分不同的数据，而这样做需要我们预先定义好数据的格式。

当我们只有一两个值的时候，这可能不是问题，但是当你有七个值的时候，你怎样一一去区分它们，而保证不出错呢？

空格不是一种很有效的方式，在我们的例子里，因为我们可以在协调装置上使用一些 JSON 库，也可以生成 JSON 格式的数据。例如下面的代码是我在处理 MPU6050 传感器时，生成 JSON 数据的代码：

```
//以 w x y z 矩阵的形式显示四元数
mpu.dmpGetQuaternion(&q, fifoBuffer);
Serial.print("{\"quat\":{ \"w\":");
Serial.print(q.w);
Serial.print(", \"x\":");
Serial.print(q.x);
Serial.print(", \"y\":");
Serial.print(q.y);
Serial.print(", \"z\":");
Serial.print(q.z);
Serial.print("}");

//用度数显示欧拉角
mpu.dmpGetQuaternion(&q, fifoBuffer);
mpu.dmpGetGravity(&gravity, &q);
mpu.dmpGetYawPitchRoll(ypr, &q, &gravity);
Serial.print(", \"position\":{ \"x\":");
Serial.print(ypr[0]);
Serial.print(";\"y\":");
Serial.print(ypr[1]);
Serial.print(",\"z\":");
Serial.print(ypr[2]);
Serial.println("}}");
```

我们并不需要很复杂的方法来处理这些数据。数据的格式是定好的，只要能生成我们想要的数据就够了。下面是最后的代码格式：

```
{
   "position": {
      "x": 0.020077044144272804,
      "y": -0.0040545957162976265,
      "z": 0.16216422617435455
```

```
    },
    "quat": {
        "w": 0.10187230259180069,
        "x": -0.02359195239841938,
        "y": -0.99427556991577148,
        "z": -0.021934293210506439
    }
}
```

我们所需要做的便是将上面传过来的字符串转换成 JSON 对象，随后我们可以不经过处理地将这些数据发送给服务器。当需要的只是 X 坐标的值时，我们直接获取 postion.x 的值就可以了。

5.4 小结

本章介绍了一些执行器、控制器、传感器，并通过一些示例对它们进行了一些简单的介绍。在这些不同的连接方式中，我们可以发现复杂的应用都是由一些简单的模式组合而来的。组合是一种很有意思的过程，但是这种过程会依据我们的需求而发生一些变化。

5.5 相关阅读资料

[1] [美] Michael Margolis. Arduino 权威指南[M]. 杨昆云译. 北京：人民邮电出版社，2015.

[2] [美] McRoberts M. Arduino 从基础到实践[M]. 杨继志，郭敬译. 北京：电子工业出版社，2013.

第6章

物联网应用示例

本章内容

- 如何可视化传感器数据
- 如何通过第三方框架来快速搭建仪表盘
- 如何快速编写一个手机程序来作为客户端

在本章中，我们将以一些实战来介绍如何延伸物联网到不同的平台中去。之前介绍的内容都以文字和数字的方式来展示数据。当只有几个数据的时候，我们可以很容易地区分它们。但是当出现大量的数据的时候，我们就无法快速地发现这些数据出现了什么问题。像温度这一类的数据，通常是缓慢变化的，趋势图可以让我们看清楚变化的趋势。甚至在我们掌握了机器学习这一类算法后，可以预测到这些值未来的变化——天气预报。值得注意的是，天气预报并不是一类神奇的算法，它是基于过去的数据而预测的。“明天降水概率 59%”这一类的预报是这样出来的：在过去的几百天里，有 59%的日子里会下雨。

对于智能系统而言，向用户展示数据可能有点多余。用户并不关心他们家的温度是多少，用户关心的是现在的温度是不是他们想要的。

在做不到如此智能的现阶段，我们可以绘制一些趋势图，向用户展示一些潜在的问题。并且，我们也已经可以让用户去控制他们的设备——通过网页，或者通过

手机。在本章中，我们还将介绍如何使用一些当前的云服务来降低开发难度。但是如果你需要考虑一些更复杂的问题，如数据安全，建议你考虑建设自己的物联网系统。

由于我们开发 APP 的技术用的是与网页相同的技术，故而数据获取等过程和浏览器端是一致的，只是换了不同的媒介而已，数据传输如图 6-1 所示。

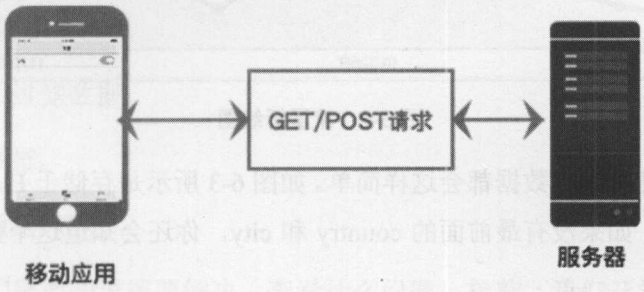


图 6-1　手机程序架构图

本章所需的软件清单：

- Android 开发环境。
- Ionic 是一款开源的 HTML 5 移动 APP 开发框架。
- Dashing 是一款开源的可视化仪表盘框架。

6.1　数据可视化

物联网应用会产生各式各样的传感器，这些传感器的值不断地注入数据库中。虽然对于一个传感器我们可能会删去它过去的值，但是对于少数的传感器，我们可能就需要一直保留它们的值，方便以后做出预测。从某种意义上来说，数据可视化更像是机器学习的上一步。机器学习的第一步也需要去解析数据，而解析数据主要依赖于人。比起数字，人们往往对于图像更为敏感，特别是在一些复杂的信息里。

如图 6-2 所示是我博客之前的流量统计图，可以看到在 10 月 30 日的时候流量突然比平时多了两倍，我就意识到这一天一定发生了什么事，然后就会去了解这一天到底发生了什么。

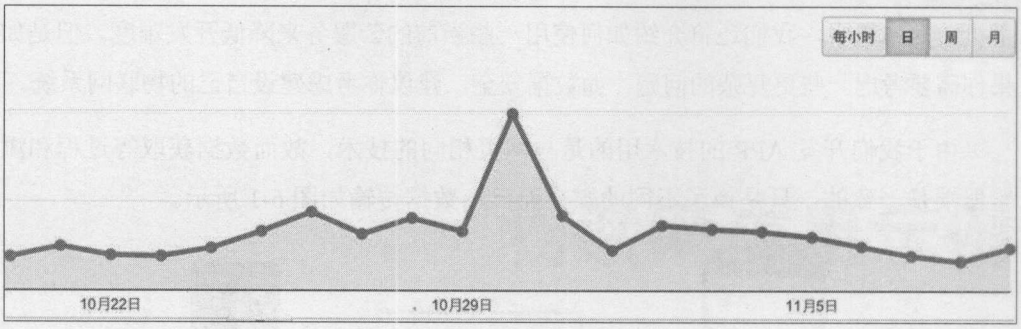


图 6-2　流量折线图

但是并不是所有的数据都会这样简单。如图 6-3 所示是存储于 ElasticSearch 中的地理位置信息，如果没有最前面的 country 和 city，你还会知道这个数字在地图上的哪个地方吗？

country	city	location	ip
China	null	105.0,35.0	210.74.157.146
China	null	105.0,35.0	210.74.157.146
China	null	105.0,35.0	210.74.157.146
China	null	105.0,35.0	210.74.157.146
China	Beijing	116.3883,39.9289	219.142.53.236
United States	Sunnyvale	-122.0074,37.4249	68.180.224.240
China	Shanghai	121.3997,31.0456	101.226.73.30
China	null	105.0,35.0	210.74.157.146
China	null	105.0,35.0	210.74.157.146
China	null	105.0,35.0	210.74.157.146
China	null	105.0,35.0	210.74.157.146
China	Guangzhou	113.25,23.1167	125.88.219.97
China	Guangzhou	113.25,23.1167	210.21.67.116
China	Zhengzhou	113.5325,34.6836	42.236.7.70
China	Fuzhou	119.3061,26.0614	117.27.153.139
China	Nanjing	118.7778,32.0617	180.96.71.233
China	Shanghai	121.3997,31.0456	180.153.211.190

图 6-3　搜索引擎中存储的地理位置信息

这时候，我们就需要了解数据可视化。

6.1.1 可视化用户数据

可视化简单来说，就是将抽象数据转换为可见图形的过程，以此来简化数据的复杂性。人们一般将数据可视化的过程分为以下几步：

（1）制定问题（可选）。

（2）收集数据。

（3）分析、过滤数据。

（4）呈现数据。

（5）挖掘数据/交互。

首先，我们需要知道需要解决、查看什么问题。接着，我们开始收集数据。通常来说，我们可以直接开始收集数据，紧接着需要了解在数据中什么是有用的数据。例如，在我们解析出 IP 信息的地理位置后，图 6-2 中的 IP 信息已经没用了。随后，我们就可以用图形的方式来呈现这些数据。如图 6-4 所示是我对自己博客过去一年的流量来源进行的统计。

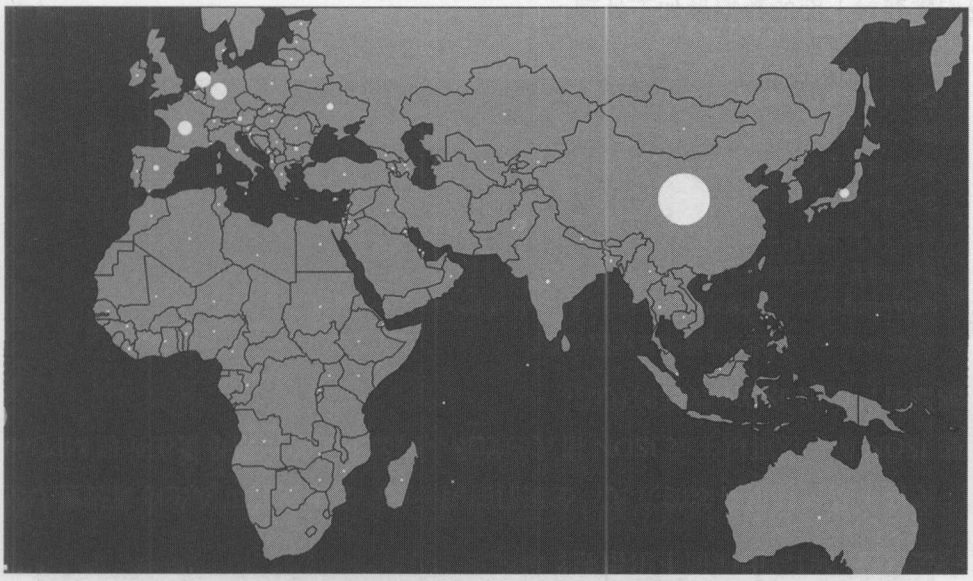


图 6-4　网站访客来源

从图 6-4 中可以发现，主要流量来源是中国，随后是德国、法国、荷兰这些欧洲国家。这样我们就可以将主要精力放在中国——这是我们在可视化这些数据后得到的结论。

这些数据原先以某种形式存储在 log 文件中，原始的信息只有 IP 信息，如下所示：

```
...
120.204.200.21 - - [10/Nov/2015:05:14:02 +0000] "GET / HTTP/1.1" 200 2955
"-" "DNSPod-Monitor/2.0" -
36.103.158.156 - - [10/Nov/2015:05:14:05 +0000] "POST
/ngx_pagespeed_beacon?url=https%3A%2F%2Fwww.phodal.com%2Fblog%2Fevery-programm
er-should-know-how-seo%2F HTTP/1.1" 204 303
"https://www.phodal.com/blog/every-programmer-should-know-how-seo/"
"Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/537.36 (KHTML, like Gecko)
Chrome/45.0.2454.85 Safari/537.36" -
180.76.15.30 - - [10/Nov/2015:05:14:24 +0000] "GET
/blog/cheng-xu-yuan-de-meng-xiang/ HTTP/1.1" 200 4306 "-" "Mozilla/5.0
(compatible; Baiduspider/2.0; +http://www.baidu.com/search/spider.html)" -
...
```

我们需要分析这些数据，并且从中过滤有用的信息，再以图形的形式表述出来。我们需要对上面的数据做如下处理：

（1）从 IP 获取地理位置信息。

（2）转换成相应的数据模式。

（3）存储数据。

在这个例子中，由于数据有 1.5GB 左右，因而使用普通的脚本处理可能难度比较大。因此，先用了 Jython 与 Hadoop 来解析这些数据，数据的格式如下所示：

```
{
"bytes": 31608,
"city": null,
"country": "China",
"date": "2014-09-22T10:33:54.000+08:00",
"ip": "210.74.157.146",
"location": "105.0,35.0",
```

```
    "referrer": "https://www.phodal.com/",
    "status": 200,
    "url": "GET /blog/ HTTP/1.1",
    "useragent": "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_10_0)
AppleWebKit/537.36 (KHTML, like Gecko) Chrome/37.0.2062.122 Safari/537.36"
    }
```

随后将这些数据存储到 ElasticSearch 中，然后我们可以通过相应的 Query 向 ElastiSearch 查询，并返回相应的结果。下面便是 ElasticSearch 发生的 Query：

```
    {"query":{"query_string":{"query":"*"}},"aggs":{"2":{"terms":{"field":"cou
ntry","size":200,"order":{"_count":"desc"}}}}}
```

不过我们要处理的数据并不会这么复杂，而且可以用一些现成的工具来帮助我们完成这项工作。

6.1.2 仪表盘

仪表盘可以说是集合了一些数据可视化模板的工具，当然也可以直接在上面显示相关的数据。它们可以让我们实时看到一些设备的状态，以及传感器值的变化，甚至可以进行交互。在这些工具中，比较流行的有 Freeboard、Dashing，其截图分别如图 6-5 和图 6-6 所示。

图 6-5 Freeboard 截图

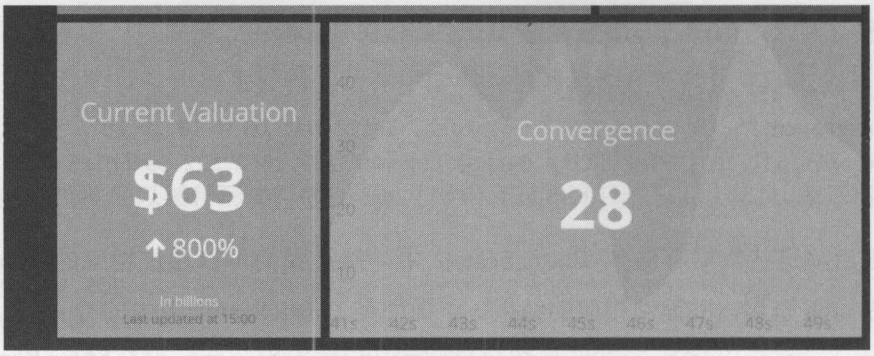


图 6-6　Dashing 截图

在这些应用上会有不同用途的不同组件。如在 Freeboard 的截图中，温度使用的是数字。它们做的事情是类似的，可以通过动态数据绑定来实时改变数据。因为它们都是开源框架，因而当这些组件不能满足我们的要求时，我们也可以制作一些相应的插件来完成我们的工作。

如果有一些数据需要更多的交互和更强的图形表现力，则可以试试使用 Processing 或者 D3.js。两者都是非常不错的工具。Processing 可以直接通过串口读取传感器的值——如果你在设计一个结合传感器值的可视化软件，如观察 MPU6050 这类的传感器的运动变化，那么这个工具会很有帮助。D3.js 则是一个 JavaScript 上的数据可视化框架，尽管很强大，但是也很复杂。

6.2　仪表盘类型示例：温度趋势图

一个简单的仪表盘可以只有一个数值，但是一个数值通常很难表达什么。用折线图来描绘温度趋势是一个非常不错的选择，我们可以了解一天的天气情况，如图 6-7 所示是百度搜索天气时返回的趋势图。

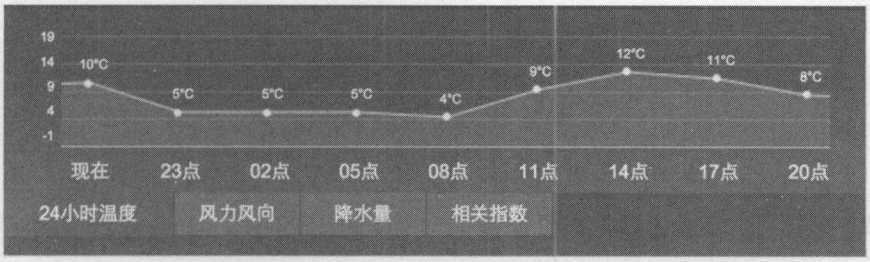


图 6-7　百度显示的温度趋势图

现在我们要做的就是创建这样一个趋势图。这一类的图表库中做得比较好的有百度的 ECharts，以及国外的 HighCharts，两者都是很不错的数据可视化工具。当然，ECharts 会有比较详细的中文文档，但是对于当前这个例子来说，可能有点大材小用。

1. 创建趋势图

首先，我们需要在服务器代码中添加一个新的页面来展示这个新的页面——这个新页面就直接称为 dashboard。先向我们的 app.js 中添加一个新的路由，它直接返回这样的一个 HTML 文件：

```
app.get('/dashboard', function (req, res) {
    'use strict';
    res.render('dashboard', {
        title: 'Dashboard'
    });
});
```

在这个 HTML 文件中，我们需要引入相应的库，需要到官方（http://echarts.baidu.com/）下载相应的库，可以使用 bower 这样的工具来安装。安装后添加相应的库进入 jade 文件中，我们的 dashboard.jade 文件结果如下所示：

```
doctype html
html
    head
        title= title
```

```
        script(src='./scripts/echarts-all.js')
        script(src='./scripts/echarts-example.js')
    body
        div(id="main" style="min-width: 310px; height: 400px; margin: 0
auto")
```

我们引入了 Echarts 库，并且创建了一个新的文件叫作 echarts-example，用来编写相应的代码。这两个文件需要放在 pubilc/scripts 目录下，其中 echarts-example 的内容如下所示：

```
window.onload = function(){
    var chart = echarts.init(document.getElementById('main'));

    var option = {
        legend: {                    // 图例配置
            padding: 5,              // 图例内边距，单位 px，默认上下左右内边距为 5
            itemGap: 10,             // Legend 各个 item 之间的间隔，横向布局时为水平
                                     // 间隔，纵向布局时为纵向间隔
            data: ['月平均气温']      // 数据标题
        },
        tooltip: {                   // 气泡提示配置
            trigger: 'item'          // 触发类型，默认数据触发，可选为：'axis'
        },
        xAxis: [                     // 直角坐标系中横轴数组
            {
                type: 'category',    // 坐标轴类型，横轴默认为类目轴，数值轴则参考 yAxis
                                     // 说明
                data: ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug',
'Sep', 'Oct', 'Nov', 'Dec']
            }
        ],
        yAxis: [                     // 直角坐标系中纵轴数组
            {
                type: 'value',       // 坐标轴类型，纵轴默认为数值轴，类目轴则参考 xAxis
                                     // 说明
                boundaryGap: [0.1, 0.1], // 坐标轴两端空白策略，数组内数值代表百
                                         // 分比
                splitNumber: 4       // 数值轴用，分割段数，默认为 5
            }
        ],
```

```
        series: [
            {
                name: '气温',      // 系列名称
                type: 'line',      // 图表类型，折线图 line、散点图 scatter、柱状图
                                   // bar、饼图 pie、雷达图 radar
                data: [7.0, 6.9, 9.5, 14.5, 18.2, 21.5, 25.2, 26.5, 23.3, 18.3, 13.9, 9.6]
            }]
    };

    chart.setOption(option);
};
```

当页面加载完后，我们会通过 document.getElementById('main')方法来找到页面中的文档元素，并且将其作为参数初始化 Echarts。接着声明了 Option 变量，并且将这个变量的值赋予 chart 对象，结果如图 6-8 所示。

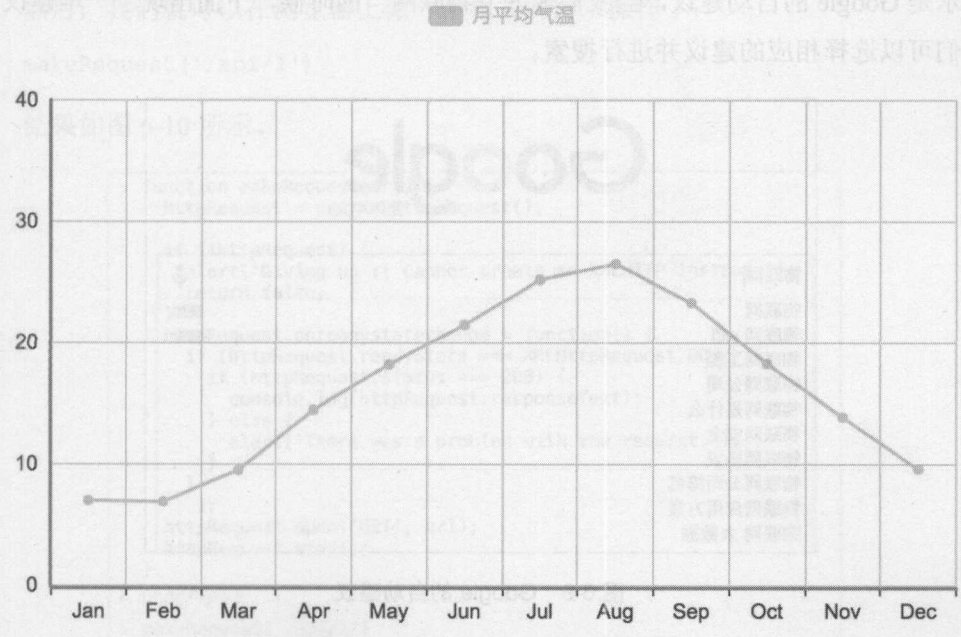


图 6-8 ECharts 温度趋势

你可能发现了，其主要内容是在 option 这个变量，其中包含了 legend、tooltip、xAxis、yAxis、series 五个对象。legend 放置的是一些图例相关的配置，tooltip 则是

当鼠标放在图上时显示的相应提醒。xAxis 则是 *X* 轴显示的内容，yAxis 是 *Y* 轴显示的内容。series 则是显示数据相关的，type 会指定图表显示的数型，这里用的是折线图，这些具体的配置可以参照官方的文档。我们主要关心的是 series 对象中的 data，这里放置的值是真实数据。

我们可以通过扩展 API 获取真实的值，并用这些值替换上面假的数据。

2. 获取最新数据

为了在网页上获取最新的数据，同时不刷新页面，这里我们需要使用一种叫作 Ajax 的技术，它的全称是 Asynchronous JavaScript and XML，即异步的 JavaScript 与 XML 技术。它可以向服务器发送并取回必需的数据，现有的主流网站都采用了这项技术。当我们在这些网站上进行搜索或输入的时候，会有一些自动建议。如图 6-9 所示是 Google 的自动建议，当我们输入“物联网”的时候，下面出现了一堆建议，我们可以选择相应的建议并进行搜索。

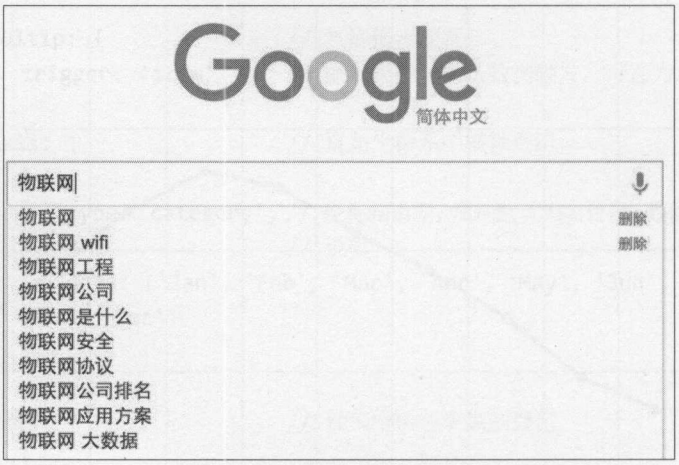


图 6-9　Google 的自动建议

除此之外，那些无限滚动的页面也大多使用了这项技术。

```
function makeRequest(url) {
  httpRequest = new XMLHttpRequest();
```

```
  if (!httpRequest) {
    alert('Giving up :( Cannot create an XMLHTTP instance');
    return false;
  }
  httpRequest.onreadystatechange = function() {
    if (httpRequest.readyState === XMLHttpRequest.DONE) {
      if (httpRequest.status === 200) {
        console.log(httpRequest.responseText);
      } else {
        alert('There was a problem with the request.');
      }
    }
     };
  httpRequest.open('GET', url);
  httpRequest.send();
}
```

然后，我们就可以在浏览器上做一个相关的请求操作了：

```
makeRequest('/api/1')
```

结果如图 6-10 所示。

```
> function makeRequest(url) {
    httpRequest = new XMLHttpRequest();

    if (!httpRequest) {
      alert('Giving up :( Cannot create an XMLHTTP instance');
      return false;
    }
    httpRequest.onreadystatechange = function() {
      if (httpRequest.readyState === XMLHttpRequest.DONE) {
        if (httpRequest.status === 200) {
          console.log(httpRequest.responseText);
        } else {
          alert('There was a problem with the request.');
        }
      }
       };
    httpRequest.open('GET', url);
    httpRequest.send();
  }
< undefined
> makeRequest('/api/1')
< undefined
  [{"_id":"56418b06bcfa06f7ab9fa022","user":"1","led":true}]
>
```

图 6-10 浏览器 Console 执行的结果

在这里，我们就可以引入 jQuery 库，这是一个跨浏览器的 JavaScript 工具库。我们可以用它来完成操作 DOM 对象、创建动画效果、事件处理、Ajax 等工作。现在，我们可以先使用它的 Ajax 功能。我们只需到官网（https://jquery.com/download/）下载相应的库，并将其添加到 HTML 文件中即可，如下所示：

```
<script src="./scripts/jquery-2.1.4.min.js"></script>
```

下面是这个库做 get 请求的例子：

```
$.get("/api/1", function(result){
   console.log(result)
});
```

下面的代码是向“/api/2”中上传（post）数据{led: true}的例子：

```
$.post( "/api/2", {led: true}, function( data ) {
  console.log(data)
});
```

整个过程如图 6-11 所示。

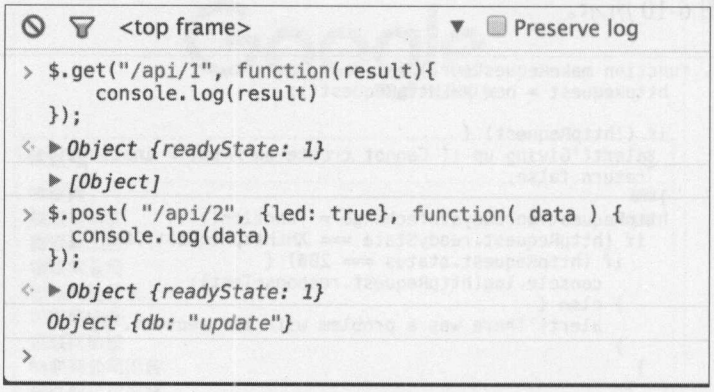


图 6-11　get/post 数据

3. 扩展 API

之前我们一直犯的一个错误便是，没有设计好 RESTful API，而这也需要一步步演进过来。如果我们只有一个用户和多个设备，那么就不需要区分这些用户——尽管我们可以为未来而设计，但是未来总是在变化的。如果我们只需要支持一个用户

的多个设备，那么为什么不把花在支持多个用户上的精力用来做一些更好的硬件设备呢？

现在，让我们重新设计一些新的 URL：

```
/user/:user_id/device/:device_id
```

举个例子，/user/1/device/2，其含义是第一个用户的第二个设备。当我们向这个 URL 上传数据的时候，应该会直接插入相应的数据。这样，就可以通过下面的 URL 来查看这个设备过去的所有结果了，而这个结果则是用于当前的 dashboard：

```
/user/:user_id/devices/:device_id/results
```

如果我们想看某个设备的第几个结果的时候，就可以通过下面的 URL 来查看，如/user/2/devices/3/result/4 可以表示第二个用户的第三个设备的第四个结果：

```
/user/:user_id/devices/:device_id/result/:result_id
```

现在我们已经可以支持多个设备了，以后只需加上一些相应的用户授权、认证和 Token 的机制就是一个简化的云服务了。

与之前的代码类似，我们只需解析出 URL 的参数，就可以辨识这些用户、设备、结果，并且在数据库中进行操作：

```
app.get('/user/:user_id/devices/:device_id/results', function (req, res) {
    var payload = {user: parseInt(req.params.user_id), device: parseInt
(req.params.device_id)};
    db.find(payload, function (results) {
        return res.json(results);
    });
});

app.post('/user/:user_id/devices/:device_id', function (req, res) {
    var data = req.body;
    data.user = parseInt(req.params.user_id);
    data.device = parseInt(req.params.device_id);

    db.insert(data);
    res.send({db: "insert"});
});
```

```
    app.get('/user/:user_id/devices/:device_id/result/:result_id', function
(req, res) {
        var payload = {user: parseInt(req.params.user_id), device:
parseInt(req. params.device_id)};
        db.findOrder(payload, parseInt(req.params.result_id), function
(results) {
            return res.json(results);
        });
    });
```

你或许注意到了上面有一个 db.findOrder 方法，这意味着我们重写了一个新的数据库操作函数，这个函数用于返回结果中的第几个结果，其代码如下所示：

```
    MongoPersistence.prototype.findOrder = function (queryOptions, order,
queryCB) {
        'use strict';
        MongoClient.connect(url, function (err, db) {
            var findDocuments = function (db, query, callback) {
                var collection = db.collection("documents");

collection.find(query).limit(1).skip(order).toArray(function (err, docs) {
                    callback(docs);
                });
            };

            findDocuments(db, queryOptions, function (result) {
                db.close();
                queryCB(result);
            });
        });
    };
```

尽管/user/:user_id/devices/:device_id/result/:result_id 这个 URL 的出现可能不是很有必要，但是在调试的过程中会很有帮助。然后向数据库中添加一些新的数据或者直接连上传感器来获取新的数据：

```
    curl -X POST -d '{ "led": true,"temperature": 15 }' -H "Content-Type:
application/json" http://localhost:3000/user/1/devices/1
```

应用上面的获取数据的方法，我们的 dashboard 代码就变成了：

```
$(function () {
    var data = [];
    $.get('/user/1/devices/1/results', function(response){
        $.each(response, function(key, val) {
            data.push(val.temperature);
        });
        var myChart = echarts.init(document.getElementById('main'));

        var option = {
            ...
        };

        myChart.setOption(option);
    });
});
```

我们先获取第一个用户的第二个设备，然后取出结果中的 temperature 的值添加到 data 数组中，再渲染出页面。

6.2.1 移动设备上查看

过去的经验告诉我，如果使用的库不支持一些新的特性，那么选择的时候需要特别谨慎。通常这会给后期带来很多麻烦，严重的话，可能需要重写这部分代码。这里由于使用的库本身已经可以很好地支持响应式的设计，因此不需要花费太多的时间去确保这部分内容是否可以在移动设备上使用。

现有的 Web 应用都需要考虑到用户可能使用着不同屏幕大小的设备。常见的有如图 6-12 所示的电脑、平板、手机，在这些设备上又有着不同的屏幕大小。

在一些常见的元素上，如标题、菜单、数值等，我们需要考虑使它们在小的设备上都是可见的。而一些特定的元素，我们可以考虑将之隐藏。这些是按情况而定的——在一些特殊的情况下，我们可能会构建出几个不同的版本，如桌面版、移动版、APP 版，这也是主流网站的一些做法。以淘宝为例，我们有桌面版的淘宝、手机版的淘宝，以及 iOS、Android、Windows Phone 版的淘宝等。

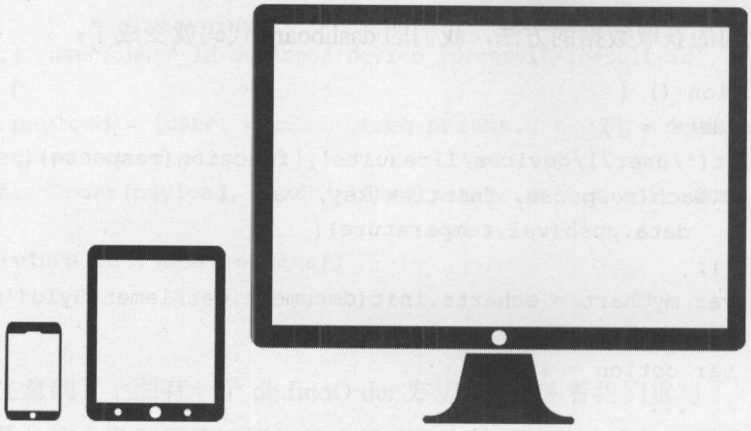


图 6-12　电脑、平板、手机设备

依据不同的需求，我们应该在不同的时期开发出不同的版本。首先，需要确认我们的用户主要使用的会是哪个版本。如果用户主要使用的是 APP，那么我们就要先开发出 APP 版本，然后才是网页版。而由于 APP 本身需要依赖于 API 或者说 Web 服务，因而我们就需要构建一个网站后台。常见的策略是开发网站版本，不仅因为其性价比高——可以运行在这些设备上、开发成本低，而且容易找到开发人员。随后，依据不同的用户需求去开发不同的版本，做到精益求精。

在 Web 方面，比较好的实践有响应式设计，可以依据不同的屏幕大小做到适应不同屏幕的风格。而在 APP 方面，比较好的一些应用就是混合应用，将在 Web 方面的实践用到手机应用上——不同平台的手机都运行着相同的 APP，而且只需要写一份代码。

6.2.2　使用 Dashing

Dashing 是一个非常优秀的 Dashboard 框架，它包含了一些预制的组件，你也可以通过写 CSS、HTML、JavaScript 来创建自己需要的组件。并且这些组件支持动态数据绑定，即我们在页面上修改某个值，可以触发一些值发生变化，而不需要刷新页面。你甚至可以通过拖放界面来重新排列窗口小部件。由于这个框架是用 Ruby 写的，因此开始之前你需要为电脑安装 Ruby，并且版本要求是 1.9 以上。安装方法在官网（http://dashing.io/）上有详细的介绍。

通过下面的命令，我们可以安装这个框架：

```
gem install dashing
```

gem 是 Ruby 的包管理工具。接着我们就可以用这个命令来新建项目：

```
dashing new dashboard
```

这时候会复制一些相关的基本文本到项目的目录下，如图 6-13 所示。

```
fdhuang ⓒtest ~/repo dashing new dashboard
      create  dashboard
      create  dashboard/.gitignore
      create  dashboard/Gemfile
      create  dashboard/README.md
      create  dashboard/assets/fonts/fontawesome-webfont.eot
      create  dashboard/assets/fonts/fontawesome-webfont.svg
      create  dashboard/assets/fonts/fontawesome-webfont.ttf
      create  dashboard/assets/fonts/fontawesome-webfont.woff
      create  dashboard/assets/images/logo.png
      create  dashboard/assets/javascripts/application.coffee
      create  dashboard/assets/javascripts/d3-3.2.8.js
      create  dashboard/assets/javascripts/dashing.gridster.coffee
      create  dashboard/assets/javascripts/gridster/jquery.gridster.js
```

图 6-13　新建 Dashboard

然后，我们可以到 dashboard 目录用 bundle 来安装它相应的依赖：

```
bundle
```

现在，让我们来启动这个服务：

```
dashing start
```

只需打开 http://localhost:3030/，即可看到之前的界面。如果一切顺利，我们将看到如下内容：

```
Thin web server (v1.6.4 codename Gob Bluth)
Maximum connections set to 1024
Listening on 0.0.0.0:3030, CTRL+C to stop
127.0.0.1 - - [01/Dec/2015 22:08:35] "GET /sample HTTP/1.1" 200 1116
0.0026
[{:x=>1, :y=>15}, {:x=>2, :y=>15}, {:x=>3, :y=>15}, {:x=>4, :y=>15},
{:x=>5, :y=>10}, {:x=>6, :y=>15}, {:x=>7, :y=>15}, {:x=>8, :y=>16}]
127.0.0.1 - - [01/Dec/2015 22:08:37] "GET /views/graph.html HTTP/1.1"
```

```
200 167 0.0007
    127.0.0.1 - - [01/Dec/2015 22:08:37] "GET /sample HTTP/1.1" 200 1116
0.0017
    127.0.0.1 - - [01/Dec/2015 22:08:37] "GET /events HTTP/1.1" 200 - 0.1370
```

下面，我们就简要介绍一下 Dashing 创建的项目的目录结构，以便我们更好地使用它。

1. 目录结构

通常来说，项目的目录会包含下面几部分：

```
.
├── Gemfile
├── Gemfile.lock
├── assets
│   ├── fonts
│   ├── images
│   ├── javascripts
│   └── stylesheets
├── config.ru
├── dashboards
│   ├── layout.erb
│   └── sampletv.erb
├── history.yml
├── jobs
│   ├── convergence.rb
│   └── twitter.rb
├── lib
├── public
│   ├── 404.html
│   └── favicon.ico
└── widgets
```

Gemfile 和 Gemfile.lock 分别是 Ruby 语言用于描述依赖的文件及其锁定依赖版本的文件——lock。Assets 字如其名——资源，里面会放置一些前端相关的 Web 资源，如 fonts（字体）、images（图像）、JavaScript 和 StyleSheets（CSS）。config.ru 则是用于这个 Web 服务的配置文件。dashboards 目录下放置的是布局相关的文件，如 layout.erb。history.yml 会保存一些历史相关的值。jobs 则是功能函数所在的地方，我们会在这个

目录下做一些获取数据的工作。lib 会存放一些库相关的文件。public 会存放一些相关的资源文件，如网站图标。widgets 目录则是一些组件，如时钟、图形等。

我们可以主要集中于 jobs、dashboards 目录，即获取数据和显示数据相关的布局。

2. 一个定时的 job

现在，先让我们来看看一个简单的 job 是怎样的。下面的代码是官方给的示例中的代码——**sample.rb** 的一部分：

```
SCHEDULER.every '2s' do
  send_event('synergy',   { value: rand(100) })
end
```

第一行代码 **SCHEDULER.every ‘2s’ do** 表明了它会每 2 秒钟执行一次下面的代码段中的内容。它将发送一个事件名为 synergy、值为 100 以内的随机数的事件。这个事件会将 API 的形式暴露给前台。在前台的模板文件 **sample.erb** 中，我们只需要有相应的 data-id 就可以在前台接收这些值。

```
      <li data-row="1" data-col="1" data-sizex="1" data-sizey="1">
        <div data-id="synergy" data-view="Meter" data-title="Synergy"
data-min="0" data-max="100"></div>
      </li>
```

然后将其显示到界面上，如图 6-14 所示。

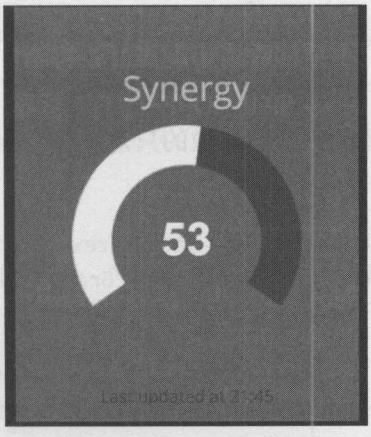


图 6-14　Dashing 示例值

同我们看到的一样，图中的 Synergy 是来自上面代码中的 data-title。而这个界面的样式则来自于 data-view，即目录中的 widgets。上面的 data-min 和 data-max 则定义了这个值的范围。

3. 整合 Dashing

我们所要做的事情和上面的差不多，即：

（1）获取我们的 API 的数据。

（2）将 API 的数据显示到前台。

最后的效果应该如图 6-15 所示。

图 6-15　Dashing 温度显示效果

效果看上去很复杂，而我们所需要做的只是：将 data-view 修改为 Graph，再修改下 title、id 就可以了。

```
    <li data-row="1" data-col="1" data-sizex="2" data-sizey="1">
      <div data-id="tempdata" data-view="Graph" data-title="温度趋势" style=
"background-color:#ff9618"></div>
    </li>
```

接着，我们可以写一个 job 来获取 API 的值，并发现相应的温度事件。这里，我们需要使用 Ruby 语言中的 http 库和 uri 库来获取数据，并用 JSON 来解析它们——这

和我们在前面用 Python 写的 API 请求是一致的，先获取数据，再解析，最后显示：

```
require "json"
require "net/http"
require "uri"

def get_data
  uri = URI.parse("http://localhost:3000/user/1/devices/1/results")

  http = Net::HTTP.new(uri.host,uri.port)
  request = Net::HTTP::Get.new(uri.request_uri)

  response=http.request(request)
  res=JSON.parse(response.body)

  index = 0
  result = []
  res.map do |data|
    index = index + 1
    result.push({x: index , y: data["temperature"].to_i})
  end
  return result
end
```

在上面代码的第 6~11 行中，我们做的事情是去获取第一个用户的第一个设备，即我们的温度传感器的数据，并将 parse 中响应的内容保存到 result 变量中。

接着，我们定义了一个 index 变量和一个名为 result 的空数组。通过 map 函数我们可以取出 res 数组中的一个个值，如下所示，并对每一个值进行单独的处理——取出其中的温度值，用 to_i 将其转换为整数。最后将一个结果放入 result 数组中，再返回 result：

```
javascript {    "_id": "5645dba5fb5b5ef43f3b6264",    "date":
"2015-11-13T12:46:29.227Z",    "device": 1,    "led": true,
"temperature": 16,    "user": 1 }
```

最后，我们在定时任务中调用这个方法：

```
SCHEDULER.every '2s' do
  points=get_data
```

```
  send_event('tempdata', points: points)
end
```

这样我们就可以得到如图 6-15 所示的结果了。

尽管使用 Dashing 会比自己创建趋势图方便，但是这类框架会限制我们的使用——这里它使用的是 Ruby 语言，这是一门比较小众的语言，并且相比于其他语言而言，可能会显得难以维护。因而在使用的时候我们需要考虑是否有人可以做相关的事情。

6.3 创建手机应用

手机应用与 Web 应用开发有很多的相似之处，它们都是调用一些接口，然后渲染出页面。

- 原生应用。原生应用是指专为特定操作系统开发的应用。这些应用可以直接访问手机的所有功能，如摄像头、蓝牙、WiFi 等。这些应用通常速度更快、性能更好。由于其直接访问系统的 API，因此性能上与混合应用相比会更好。但是这里有一个问题——需要支持开发的设备太多，开发成本由此升了上去。
- Web 应用。Web 应用是指运行于浏览器上的应用。Web 应用就不存在开发成本高的问题，一次开发就可以在桌面、移动浏览器上运行。然而，Web 应用对网速的要求比较高，并且与原生应用相比，用户体验不好。尽管 HTML 5 可以解决一些问题，但是这些问题还是很明显。
- 混合应用。混合应用是原生应用和 Web 应用的结合体。从技术的角度来说，混合应用就是调用浏览器，即 WebView，来运行 Web 代码。而它不仅仅是 Web 应用的离线版，它还可以通过一些框架，如 Cordova，直接调用系统的 API。在一些框架中，它甚至可以用封装系统的 UI 组件，以 Web 常用的形式来提供 API。而在混合应用框架中，可能并没有包含所有的功能，这时候就需要自己去实现。

选择哪种应用来作为用户界面，应该取决于是否有充足的时间、精力和人员。Web 应用通常更容易开发，并且直接可以在浏览器上运行。而混合应用和原生应用

往往是更好的选择，它们有着更好的用户体验，以及更快的速度。

在这些平台上，通常它们的接口都是相似的，但是选择一些平台可能会导致不适合于另外一个平台的用户进行实战。笔者之前在学习物联网系统设计的时候，也开发过 Android 平台的 APP，详情见 https://github.com/iot-works/iot-android。

在这里我们以混合应用为例，不仅可以满足多数人的需求，而且可以降低学习成本——用 JavaScript 编写代码。因为不同平台所使用的语言不同，如 iOS 用的是 Swift、Objective-C，Android 用的是 Java，Windows Phone 用的是 C#、JavaScript，这就需要学习不同的语言才能开发。

6.3.1 Ionic 简介

Cordova 是一个开源的移动设备开发框架，它提供了一组设备相关的 API，通过这组 API，移动应用能够以 JavaScript 访问原生的设备功能，如摄像头、麦克风等。

而 Ionic 是一个基于 Cordova 的、使用 HTML 5 构建移动应用的高级开发框架，它自称是“Navtie 与 HTML 5 的结合”。这个框架提供了很多基本的移动用户界面范例，如列表、标签栏和触发开关等，并提供了一些复杂的可视化布局示例，如滑出式菜单。

与基于 Cordova 框架的应用相比，它具有下面一些优点：

- 性能比用 jQuery 构建好。
- 基于 Angluar JS。
- 原生化。
- 设计精美。
- 学习乐趣。
- 为极客构建。

Ionic 遵循视图控制模式，易于理解，并与 Cocoa 触摸框架类似。在操作之前我

们需要先安装这个框架：

```
npm install -g cordova ionic
```

在安装期间，如果遇到一些网络问题，可以试着用下面的命令，使用淘宝的软件源：

```
npm config set registry https://registry.npm.taobao.org
```

安装完成后，我们就可以使用 Ionic 来创建项目了。我们可以用下面的命令来创建一个名为 iotApp 的应用：

```
ionic start iotApp tabs
```

在这个过程中，它会从 Github 上下载已经存在的项目脚手架。创建及安装过程如图 6-16 所示。

```
fdhuang @test ~/repo ionic start iotApp tabs
Creating Ionic app in folder /Users/fdhuang/repo/iotApp based on tabs project
Downloading: https://github.com/driftyco/ionic-app-base/archive/master.zip
[============================]  100%  0.0s
Downloading: https://github.com/driftyco/ionic-starter-tabs/archive/master.zip
[============================]  100%  0.0s
Updated the hooks directory to have execute permissions
Update Config.xml
Initializing cordova project
Adding in iOS application by default
Added ios platform

Your Ionic project is ready to go! Some quick tips:
```

图 6-16　Ionic 安装过程图 1

同时，还有一些提示，如图 6-17 所示。

现在，我们需要到这个目录中执行下面的命令，来搭建 SASS 环境：

```
ionic setup sass
```

然后，可以执行下面的命令来启动应用：

```
ionic serve
```

最后，你会在浏览器中看到类似的界面，如图 6-18 所示是该应用运行在 iOS 上的截图。

```
Your Ionic project is ready to go! Some quick tips:

 * cd into your project: $ cd iotApp

 * Setup this project to use Sass: ionic setup sass

 * Develop in the browser with live reload: ionic serve

 * Add a platform (ios or Android): ionic platform add ios [android]
   Note: iOS development requires OS X currently
   See the Android Platform Guide for full Android installation instructions:
   https://cordova.apache.org/docs/en/edge/guide_platforms_android_index.md.html

 * Build your app: ionic build <PLATFORM>

 * Simulate your app: ionic emulate <PLATFORM>

 * Run your app on a device: ionic run <PLATFORM>

 * Package an app using Ionic package service: ionic package <MODE> <PLATFORM>
```

图 6-17　Ionic 安装过程图 2

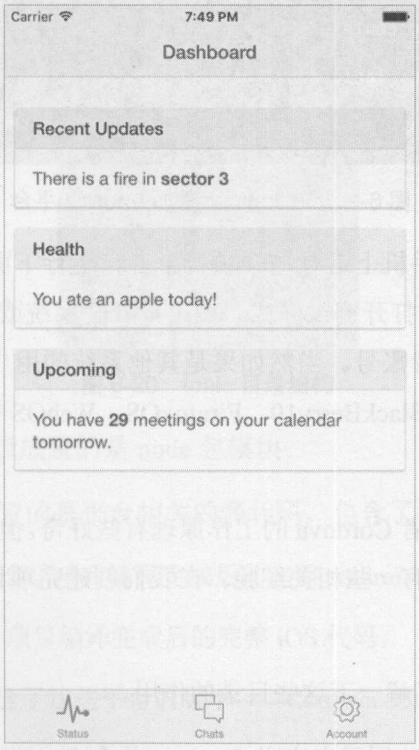


图 6-18　iOS 上的截图

为了让它可以在手机上运行，我们需要先添加这个平台：

```
ionic platform add ios
```

如果你使用的也是 Android 手机、Nokia Lumia 手机，那么你需要把上面的 iOS 改成 Android，或者是 WP8。图 6-19 是添加 Android 平台的过程。

```
Your Ionic project is ready to go! Some quick tips:

 * cd into your project: $ cd iotApp

 * Setup this project to use Sass: ionic setup sass

 * Develop in the browser with live reload: ionic serve

 * Add a platform (ios or Android): ionic platform add ios [android]
   Note: iOS development requires OS X currently
   See the Android Platform Guide for full Android installation instructions:
   https://cordova.apache.org/docs/en/edge/guide_platforms_android_index.md.html

 * Build your app: ionic build <PLATFORM>

 * Simulate your app: ionic emulate <PLATFORM>

 * Run your app on a device: ionic run <PLATFORM>

 * Package an app using Ionic package service: ionic package <MODE> <PLATFORM>
```

图 6-19　在 Ionic 上添加 Android 平台

现在，我们可以在手机上运行这个项目了。在运行下面的代码之前，你需要确定你的 Android 手机已经打开调试模式。如果是 iOS 系统或者 Windows Phone 系统，则请确认你已经有开发者账号。当然如果是其他系统的用户，也不需要担心，Ionic 还支持 Amazon-Fireos、BlackBerry10、FirefoxOS、WebOS 等操作系统。

```
ionic run android
```

你可能会对 Ionic 或者 Cordova 的工作原理有些好奇，但是如果你开发过 Android 应用或者 Web 应用就会有一些相关经验。在我们创建完项目后，项目目录结构如图 6-20 所示。

下面我们可以详细了解一下这些目录的作用。

hook 目录包含了一些用于自定义 Cordova 的命令，按照不同的顺序可以执行不

同的事情，如 after_build、before_build，即在构建之前和构建之后做的事。

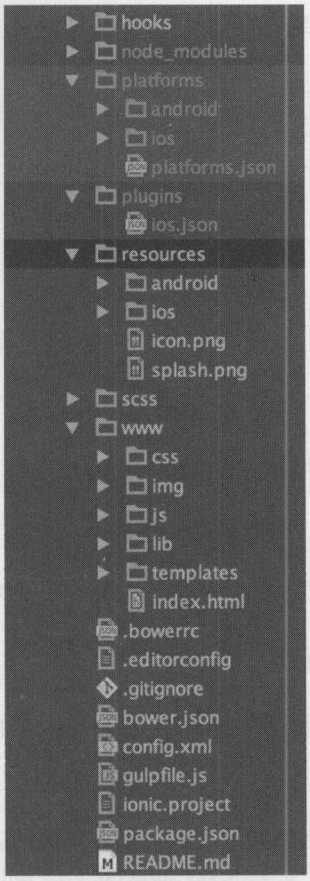


图 6-20　Ionic 目录结构

`node_modules` 目录放置的是 node 包模块。

`platforms` 目录放置的是平台相关的源代码，包含了下面的文件夹：

- android 目录下包含了项目编译完成后的完整 Android 代码。
- ios 目录下包含了项目编译完成后的完整 iOS 代码。
- platforms.json 包含了这些平台的版本——Cordova 上这些平台的版本。

`plugins` 目录会包含一些第三方插件的内容。

resources 主要用于放置资源文件，如启动界面、图标等。

scss 是一个 CSS 预处理器， 这个目录主要会包含一些样式。

www 这个目录是我们工作的目录，包含了下面的文件夹，这些文件夹名副其实：

- css
- img
- js
- lib
- templates
- index.html

.bowerrc：配置了默认的前端库安装目录。

.editorconfig：编辑器相关配置。

.gitignore：git 相关配置。

bower.json：bower 的配置，会包含需要安装的前端的名和信息，用于安装前端库。

config.xml：项目相关的配置，其中会有 APP 名称、包名、简介、分辨率等。

gulpfile.js：包含构建系统的一些命令。

ionic.project：Ionic 项目配置。

package.json：node 的包管理配置文件。

看过目录你可能知道了，我们是用一份代码构建出不同平台的代码。首先，在我们添加平台的时候，会下载相应系统的一些基础代码，同时也会下载相应平台的代码。接着，在我们运行的时候，会把 www 目录下的内容复制到一个新的目录。最后在运行的时候，会直接到这个目录下打包并运行。

而 Cordova 会充当其中的桥梁——用于连接 WebView 和原生 API。

6.3.2 趋势图

由于都是 Web 应用，因此我们可以使用 HighChart.js 作为图形绘制库。由于 Ionic 用的是 Angluar，所以我们需要先安装一个名为 ngCordova 的插件。ngCordova 是在 Cordova API 基础上封装了很多开源的 Angular.js 扩展，使开发者在 Angular.js 代码中有访问设备 API 的能力。

首先，我们需要先安装这个插件：

```
bower install ngCordova
```

然后，将下面的代码添加到 index.html 中，我们就可以在 Controller 中使用它了：

```
<script src="lib/ngCordova/dist/ng-cordova.js"></script>
```

为了在上面使用 highcharts，我们仍需要安装 jQuery、highcharts 和 highcharts-ng：

```
bower install jquery
bower install highcharts
bower install highcharts-ng
```

并添加相应的库到 index.html 文件中：

```
<script src="lib/jquery/dist/jquery.min.js"></script>
<script src="lib/highcharts/highcharts.src.js"></script>
<script src="lib/highcharts-ng/dist/highcharts-ng.js"></script>
```

现在，我们可以修改它们的代码。打开 tab-dash.html，将里面的内容换成：

```
<ion-view view-title="Dashboard">
  <ion-content class="padding">
    <div class="list card">
      <highchart id="chart1" config="chartConfig"></highchart>
    </div>
  </ion-content>
</ion-view>
```

接着，打开 controllers.js，在 DashCtrl 中添加下面的内容：

```
$scope.chartConfig = {
  title: {
    text: '月平均气温',
    x: -20 //center
  },
  xAxis: {
    categories: ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
      'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']
  },
  yAxis: {
    title: {
      text: 'Temperature (°C)'
    },
    plotLines: [{
      value: 0,
      width: 1,
      color: '#808080'
    }]
  },
  tooltip: {
    valueSuffix: '°C'
  },
  legend: {
    layout: 'vertical',
    align: 'right',
    verticalAlign: 'middle',
    borderWidth: 0
  },
  series: [{
    name: 'Today',
    data: [7.0, 6.9, 9.5, 14.5, 18.2, 21.5, 25.2, 26.5, 23.3, 18.3, 13.9,
9.6]

  }]
}
```

修改完成后，保存并运行。在 iOS 上的运行效果如图 6-21 所示。

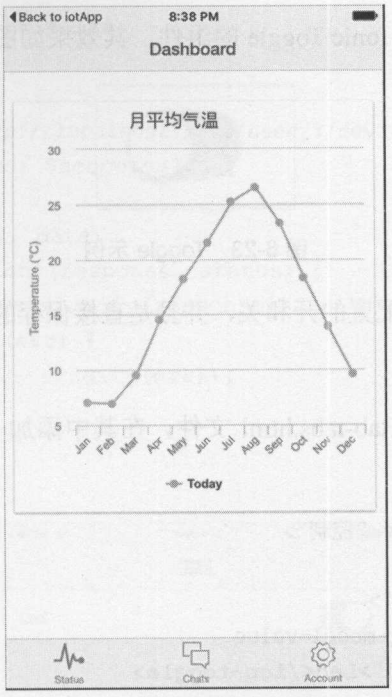


图 6-21　iOS 上的运行效果

接着，我们需要获取真实的数据，并将真实的数据渲染到页面上。

6.3.3　控制硬件

通过手机控制硬件的原理大致上相似，如图 6-22 所示。手机通过网络将数据上传到服务器，硬件（Raspberry Pi 或者 Arduino）从服务器获取状态，并通过它来操作硬件。

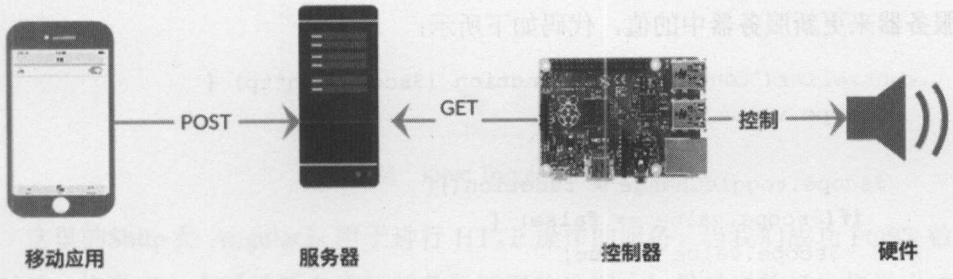


图 6-22　控制过程图

首先，我们需要添加 Ionic Toggle 的组件，其效果如图 6-23 如示。

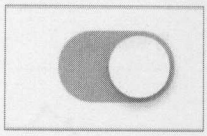


图 6-23　Toggle 示例

它可以用来控制某个配置的开和关，并且是直接保存的，即不需要我们按保存按钮。

然后，修改代码中的 tab-tabs.html 文件，在其中添加一个 ion-toggle，变成下面的代码：

```
<ion-view view-title="控制">
  <ion-content>
    <div class="list">
      <ion-toggle ng-model=value
ng-change="toggleChange()">Led</ion-toggle>
    </div>
  </ion-content>
</ion-view>
```

这里的ng-model用于配置toggle的默认值。通常在控制之前我们需要先获取LED的状态，这里我们省略了这个过程。后面的 ng-change 赋予的值是 toggleChange()，这是一个我们需要在 Controller 中添加的函数。当我们改变了开关的值的时候，将调用这个方法。修改完成后，我们的界面将如图 6-24 所示。

现在我们需要写 toggleChange 方法，在我们获取数据变化的时候将这个值 POST 给服务器来更新服务器中的值，代码如下所示：

```
.controller('ControlCtrl', function ($scope, $http) {
  $scope.value = false;

  $scope.toggleChange = function(){
    if($scope.value == false) {
      $scope.value = true;
    } else {
```

```
        $scope.value = false;
      }

      var url = 'http://localhost:3000/user/1/devices';
      var data = {led: $scope.value};

      $http.post(url, data)
       .then(function (response, status) {
         alert(JSON.stringify(response));
       }, function (err) {
         alert(JSON.stringify(err));
       });
    };
})
```

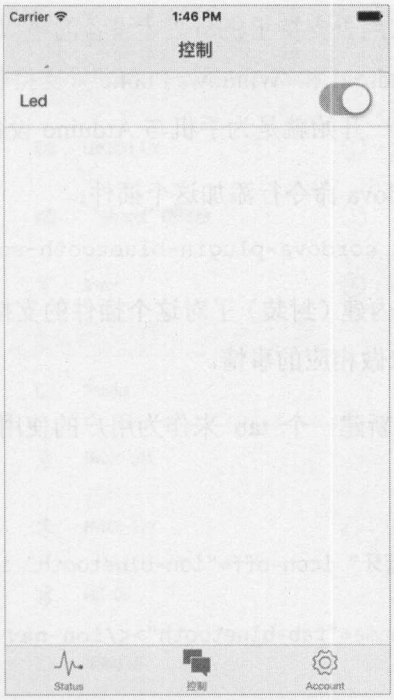


图 6-24　Ionic Toggle 效果图

这里的$http 是 Angular.js 用于进行 HTTP 操作的服务。当我们成功 POST 数据的时候，将弹出一个对话框来显示服务器返回的数据；如果错误的话，将弹出错误

信息。为了将代码用于试验，你需要修改这里的 URL，可能还需要修改对象的数据字段。

6.3.4 用蓝牙来与硬件通信

现在市场上流行的那些可穿戴式设备中，那些需要与手机连接的都具有蓝牙连接功能。手机是一个非常不错的协调装置，手机上配备着 WiFi、蓝牙，以及 2G、3G、4G 通信系统，可以直接连接硬件，并上传数据到网络上。而且由于市场上的手机应用开发已经趋于成熟，并且有众多的开发人员，使得这一类的开发工作相对比较简单。

我们仍然使用 Ionic 来开始这部分的功能，并且使用一个名为 **BluetoothSerial** 的 Cordova 蓝牙插件。它用于支持手机与蓝牙设备的串口通信，可以支持 Android、iOS、Windows Phone。Android 和 Windows Phone 只支持传统蓝牙，iOS 使用的是蓝牙低功耗，并且这个插件一开始就是为手机与 Arduino 设备通信而开发的。

现在，让我们用 Cordova 命令行添加这个插件：

```
cordova plugin add cordova-plugin-bluetooth-serial
```

由于 ngCordova 已经内建（封装）了对这个插件的支持，因此我们可以直接调用 $cordovaBluetoothSerial 来做相应的事情。

在此之前，我们需要新建一个 tab 来作为用户的使用入口，修改 tabs.html，在 ion-tabs 标签内添加：

```
    <!-- Bluetooth Tab -->
    <ion-tab title="蓝牙" icon-off="ion-bluetooth" icon-on="ion-bluetooth"
href= "#/tab/bluetooth">
      <ion-nav-view name="tab-bluetooth"></ion-nav-view>
    </ion-tab>
```

然后，在我们的 app.js 中添加相应的路由处理：

```
    .state('tab.bluetooth', {
      url: '/bluetooth',
      views: {
```

```
    'tab-bluetooth': {
      templateUrl: 'templates/tab-bluetooth.html',
      controller: 'BlueToothCtrl'
    }
  }
})
```

上面的 templateUrl 指明了我们新的模板名是 **tab-bluetooth.html**，相应的 controller 是 **BlueToothCtrl**。接着，我们需要去创建相应的模板和控制器。

在我们的模板文件中需要两个列表，一个用于显示已配对的设备，另一个显示未连接的设备。如图 6-25 所示是系统的蓝牙连接界面，图中有两种属性：已配对设备和可用设备。

图 6-25 蓝牙应用截图

我们所要做的就是创建一个这样的列表——在这个列表中，我们需要列出所有的设备，当我们单击设备的时候，就会连接到设备。一个已配对设备的模板文件代

码如下所示：

```
<ion-list>
    <div class="item item-divider">已配对</div>
    <ion-item class="item-icon-right" ng-repeat="device in listDevices"
type= "item-text-wrap"
              ng-click="connect(device.address)">
      <h2>{{device.name}}</h2>

      <p>
        {{device.id}}<br/>
      </p>
    </ion-item>
  </ion-list>
```

这里的ng-repeat便是列出所有已经列出来的设备，取出其中的每个值赋予device，再从 device 中取出设备的 MAC 地址和设置名。当我们单击设备的时候，将调用connect方法来连接设备。代码中的ng-click即为监听设备是否被单击的函数。

同理，对于未配对的设备也是如此：

```
<ion-list>
  <div class="item item-divider">未配对</div>
  <ion-item class="item-icon-right" ng-repeat="device in
discoverDevices" type="item-text-wrap"
            ng-click="connect(device.address)">
    <h2>{{device.name}}</h2>

    <p>
      {{device.id}}<br/>
    </p>
  </ion-item>
</ion-list>
```

现在，我们将调用connect方法。而在connect方法中，我们将调用$cordova-BluetoothSerial 中的 connect 方法来连接到设备。当我们连接到设备的时候，即运行到then()方法时，将读取串口的数据：

```
$scope.connect = function (address) {
```

```
    $cordovaBluetoothSerial.connect(address).then(function (err) {

      $cordovaBluetoothSerial.read().then(function (result) {
        $scope.data = result;


      })
    });
  };
```

我们可以在页面上使用一个 data 变量显示相应的值，也可以再次将这些数据发送给服务器，发送代码同之前开关 LED 一样：

```
        $http.post($localstorage.get('control_url'), {temperature:
result})
          .then(function (response, status) {
            console.log(JSON.stringify(response));
          }, function (err) {
            console.log(JSON.stringify(err));
          });
```

在那之前，我们需要一个方法来列出已配对的设备——调用 list 方法来列出现有的已配对的设备，这些设备的相关值会存储到 listDevices 变量中，而那些未配对的设备将会被存放到 discoverDevices 变量中：

```
  $cordovaBluetoothSerial.list().then(function (result) {
    $scope.listDevices = result;

    $cordovaBluetoothSerial.discoverUnpaired().then(function (result) {
      console.log("未配对", JSON.stringify(result));
      $scope.discoverDevices = result;
    }, function (err) {
      alert(err);
    });
  });
```

现在，让我们运行这个 APP，然后连接上设备试试。

如果没有意外，会出现如图 6-26 所示的效果。接着，我们就可以配对设备，以接收数据、上传数据。

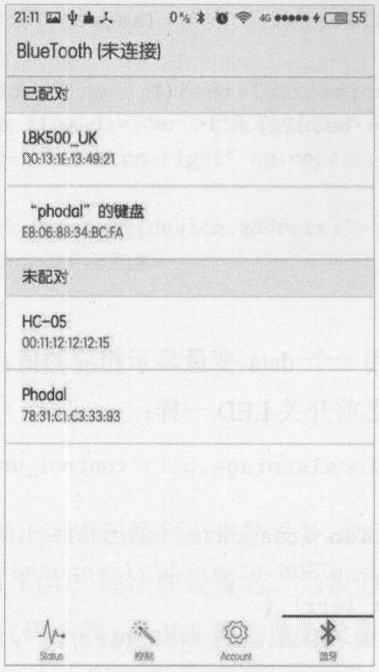


图 6-26　蓝牙界面截图

6.4　使用 AWS 云平台构建物联网

当我们没有足够的能力或者时间去开发一些物联网云平台的时候，使用一些现有的成熟的云平台是一个很好的选择。通常来说，这些平台可以为我们带来更快的开发速度和更简单的解决方案，并且可以让我们集中于实现上的业务逻辑。我们只需连接上我们的设备、上传我们的数据，就可以在地球的另一端观察这些数据的变化。

在这些物联网平台中，做得比较不错的有：

- IBM 的 Bluemix Internet of Things
- Microsoft 的 Azure IoT Suite
- Amazon 的 AWS IoT

由于亚马逊的云服务比较领先，所以这里我们以亚马逊的 AWS IoT 作为示例。

AWS IoT 是亚马逊推出的一款托管的云平台，它可以使互联设备轻松安全地与云应用程序及其他设备交互。它可以支持数十亿台设备和数万亿条消息，并且我们可以对这些消息进行处理，同时将其安全可靠地路由至 AWS 终端节点和其他设备。我们还可以结合 AWS 的其他服务，如 AWS Lambda、Amazon Kinesis、Amazon S3、Amazon Machine Learning 和 Amazon DynamoDB，来构建 IoT 应用程序，以方便我们收集、处理和分析互联设备生成的数据。

亚马逊的物联网解决方案和我们之前写的服务端代码一样，如图 6-27 所示。我们将在服务端接收设备传过来的数据并存储到服务器上，然后，通过手机等就可以直接访问这个 API，并可以对设备进行控制——这里采用的是 MQTT 协议。

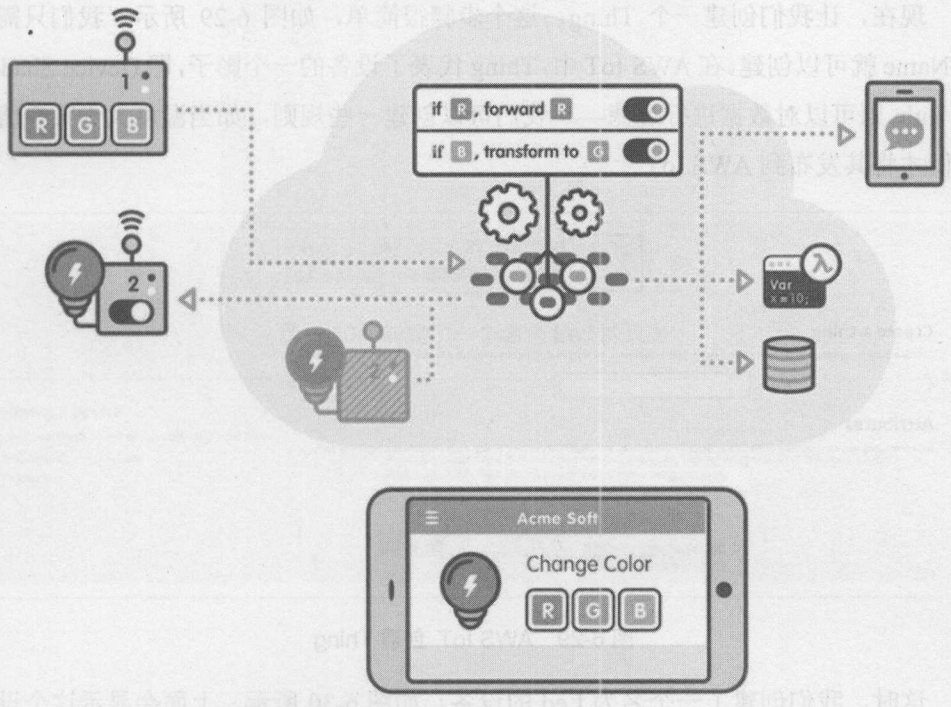


图 6-27 AWS IoT

现在让我们登录 AWS 后台，可以看到如图 6-28 所示的内容。在本书写作的时

候，AWS IoT 处于测试版。但是相信在读者看到这本书的时候，已经可以使用了。

图 6-28　AWS IoT 控制台

现在，让我们创建一个 Thing。这个步骤很简单，如图 6-29 所示，我们只需填写 Name 就可以创建。在 AWS IoT 中，Thing 代表了设备的一个影子，即 Device Shadow。而 Rule 则可以对数据进行过滤——我们可以创建一些规则，如当温度大于某个值时我们才将其发布到 AWS IoT 上。

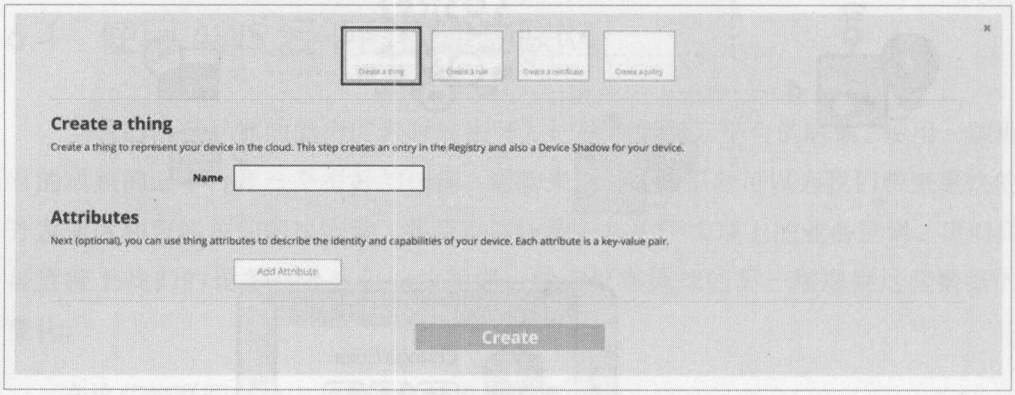


图 6-29　AWS IoT 创建 Thing

这时，我们创建了一个名为 Led 的设备，如图 6-30 所示。上面会显示这个设备的 REST API 地址，及其 MQTT Topic——我们只需订阅这个主题就可以使用这个设备了。然后单击下面的“Connect a device”按钮就会进入如图 6-31 所示的密钥配置

页，确认并下载相应的文件就可以使用了。

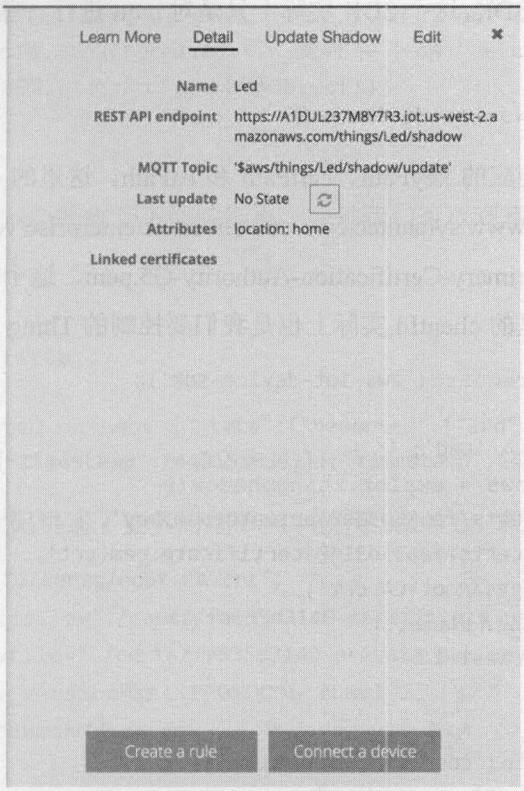


图 6-30　创建了一个名为 Led 的设备

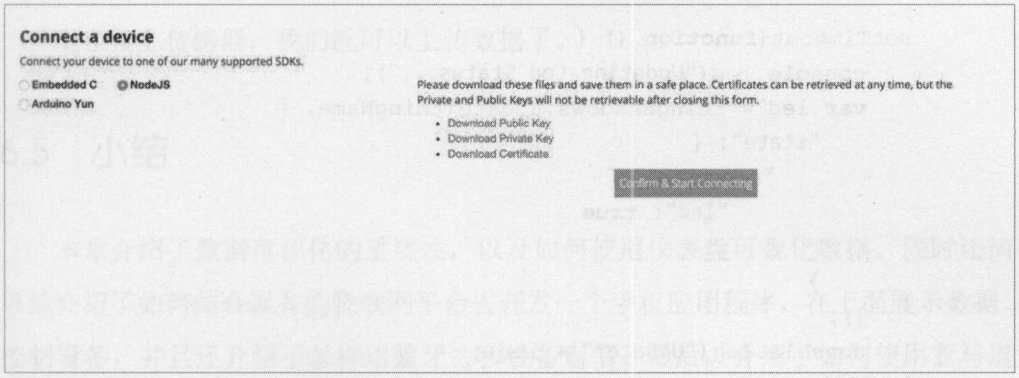


图 6-31　密钥配置页

现在，我们可以编写代码来测试我们的设备是否能正常使用了。这里我们需要使用到 AWS IoT 的 SDK，这个 SDK 实际上只是对 mqtt 进行一些封装，安装这个 SDK 的过程很简单：

```
npm install aws-iot-device-sdk
```

我们需要配置相应的 keyPath、certPath 和 caPath。这里的 caPath 需要到网上下载（网址：https://www.symantec.com/content/en/us/enterprise/verisign/roots/VeriSign-Class%203- Public-Primary-Certification-Authority-G5.pem，这个网址包含在代码里）。需要注意的是，这里的 clientId 实际上也是我们要控制的 Thing 的名字：

```
var awsIot = require('aws-iot-device-sdk');

var thingName = 'Led';
var thingShadows = awsIot.thingShadow({
   keyPath: 'certs/fa635d3140-private.pem.key',
   certPath: 'certs/fa635d3140-certificate.pem.crt',
   caPath: 'certs/root-CA.crt',
   clientId: thingName,
   region: 'us-west-2'
});

thingShadows.on('connect', function () {
   console.log("Connected...");
   thingShadows.register(thingName);

   setTimeout(function () {
      console.log("Updating Led Status...");
      var led = thingShadows.update(thingName, {
         "state": {
            "reported": {
               "led": true
            }
         }
      });
      console.log("Update:" + led);
   }, 2500);
```

```
    thingShadows.on('status',
        function (thingName, stat, clientToken, stateObject) {
            console.log('received ' + stat + ' on ' + thingName + ': ' +
                JSON.stringify(stateObject));
        });
});
```

接着运行这个脚本会接收到下面的结果，同时我们可以观察到网页上的 Shadow 值也发生了相应的变化。

```
Connected...
Updating Led Status...
Update:Led-0
received accepted on Led: {"state":{"reported":{"led":true}},"metadata":
{"reported":{"led":{"timestamp":1449151056}}},"timestamp":1449151056}
```

下面的 Python 代码讲述了更多的细节——使用 SSL/TLS 来连接 MQTT 服务器：

```
mqttc.tls_set("./certs/root-CA.crt",
              certfile="./certs/fa635d3140-certificate.pem.crt",
              keyfile="./certs/fa635d3140-private.pem.key",
              tls_version=ssl.PROTOCOL_TLSv1_2,
              ciphers=None)

mqttc.connect("a1dul237m8y7r3.iot.us-west-2.amazonaws.com", port=8883)  #
AWS IoT service hostname and portno
```

现在接上传感器，我们就可以上传数据了。

6.5 小结

本章介绍了数据可视化的重要性，以及如何使用仪表盘可视化数据。同时还简单地介绍了如何结合现有的物联网平台去开发一个手机应用程序，在上面展示数据、控制设备，并且还介绍了怎样用蓝牙去和设备通信。最后，介绍了如何使用亚马逊的 AWS IoT 来开发物联网应用。

6.6 相关阅读资料

[1] [美] Julie Steele，Noah Iliinsky. 数据可视化之美[M]. 祝洪凯，李妹芳译. 北京：机械工业出版社，2011.

[2] [挪] Magnus Lie Hetland. Python 基础教程（第 2 版•修订版）[M]. 司维，曾军崴，谭颖华译. 北京：人民邮电出版社，2014.

[3] [美] Wes McKinney. 利用 Python 进行数据分析[M]. 唐学韬译. 北京：机械工业出版社，2013.

[4] [美] Ari Lerner. AngularJS 权威教程[M]. 赵望野，徐飞，何鹏飞译. 北京：人民邮电出版社，2014.

[5] [美] Pawel Kozlowski，Peter Bacon Darwin. 精通 AngularJS[M]. 李路，王永强，马海波译. 武汉：华中科技大学出版社，2014.

第 7 章

真正的物联网：MQTT 与 CoAP 协议

本章内容

- 介绍 CoAP 协议
- 介绍 MQTT 协议
- 集成 CoAP 和 MQTT 协议到系统中

采用 HTTP 协议的物联网系统算不上真正的物联网系统。在本章中我们将介绍 MQTT、CoAP 等物联网协议，它们可以帮助我们更好地处理物联网系统中的消息通信。

HTTP 协议是一个非常不错的协议，不仅仅因为它应用广泛，还因为它有非常多的相关库支持。虽然在嵌入式系统领域也有类似于 uIP 这样的库可以支持 TCP/IP 协议，但是它还是太重了。在本章中，我们会介绍两个新的协议：**MQTT** 及 **CoAP**。

MQTT（全称：MQ Telemetry Transport）协议拥有一个比较特别的消息模式：**发布/订阅模式**。

CoAP（全称：Constrained Application Protocol）协议具有一些非常适合于嵌入式设备的特点：

- CoAP 采用了二进制报头，而不是 HTTP 协议的文本报头（HTTP 协议是基于字符的）。
- CoAP 降低了头（header）的可用选项的数量。
- CoAP 减少了一些 HTTP 的方法，删除如 TRACE、OPTIONS 这些方法。
- CoAP 可以支持检测装置。

本章我们实现的物联网系统是在第 3 章中提到过的六边形架构的实现。其架构如图 7-1 所示。

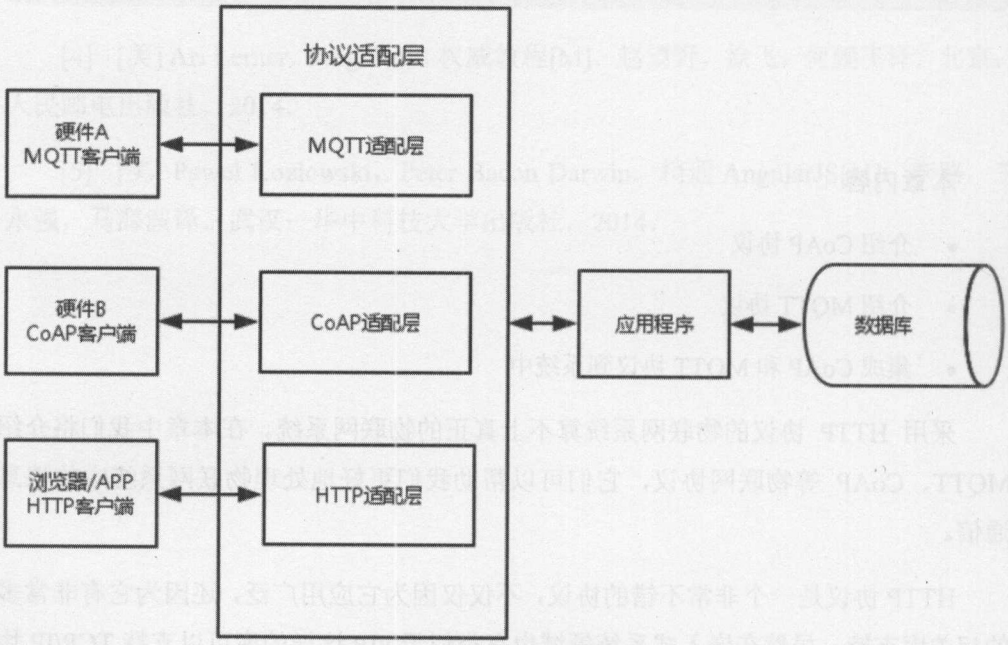


图 7-1　第 7 章架构图

结合之前的 HTTP 协议，以及这章中的 MQTT 和 CoAP 协议，我们将要适配三个协议。这三个协议间接地与数据库相连接来进行相应的数据操作。一个好的实现是：抽象出这些协议来构成上图中的应用程序。

7.1 MQTT

MQTT 是 IBM 开发的基于客户端/服务端架构的发布/订阅模式的消息传输协议。该协议支持所有平台，几乎可以将所有的联网物品和外部网络连接到一起，被用于传感器和制动器的通信协议。它的设计思想是轻巧、开放、简单、规范。当前这个协议使用最广泛的领域是 Android 应用，开发人员使用这个协议来实现 Android 客户端上的消息推送。

和我们之前讨论的 HTTP 协议不同的是，MQTT 是有序的、可靠的、双向字节流传输的。并且 MQTT 协议不再是请求/响应模式，而是发布/订阅模式。这种模式和我们在社交软件中关注某个人是类似的，当某个人的状态更新的时候，我们将收到这个人的更新状态。在这种模式中有两个主要的角色，即发布者和订阅者。我们是订阅者，我们关注的人是发布者。当发布者发布信息的时候，我们就可以接收到这些信息。订阅者知道自己订阅了相关的内容，但是发布者并不知道订阅者订阅了相关的内容。

这个协议具有以下一些特点[1]：

- 使用发布/订阅消息模式，提供了一对多的消息分发，发布者与订阅者松散地耦合。
- 消息传输不需要知道负载内容。
- 提供了三种等级的服务质量（QoS）。
- 传输消耗和协议数据交换很小，并最大限度地减少网络流量。
- 发生异常连接断开时，能通知到相关的订阅者和发布者。

7.1.1 MQTT 消息订阅示例

在这种订阅模式中，一个订阅会包含主题过滤器（Topic Filter）及最大服务质量

1 源自 http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html，主要参考了其 MQTT 3.1.1 中文版：https://github.com/mcxiaoke/mqtt。

（QoS）等级。而这里的主题指的是：附加在应用消息上的一个标签（或者说标志）。服务端需要先知道这个主题的存在，才能与已知的主题相匹配。例如，我们可以注册温度传感器的主题名是 temperature。我们只需要在客户端创建一个类似的发布者，来发布最新的传感器数据，并创建一个订阅者来接收这些消息。

现在，我们可以先看看这个协议的一些示例，然后再看看其工作原理。

mosquitto

mosquitto 是一个开源的 MQTT3.1 协议的代理服务器的实现，并由 MQTT 协议创始人之一 Andy Stanford-Clark 开发。Mosquitto 的安装方法和我们之前说到的类似，不同的操作系统和平台有不同的安装方法[2]。

在安装完 mosquitto 后，会提供下面的几个工具：

（1）mosquitto：MQTT 代码服务器。

（2）mosquitto_pub：用于发布消息的命令行客户端。

（3）mosquitto_sub：用于订阅消息的命令行客户端。

作为一个 MQTT 服务端，mosquitto 所做的实际上是订阅者与发布者的消息中介（或者说是代理）。它所需要做的就是接受来自客户端的网络连接、发布消息，并处理客户端的订阅及取消订阅的请求。同时它也将转发应用消息给符合条件的订阅客户端。

现在，我们可以运行 **mosquitto** 命令启动一个 mosquitto 服务端，启动过程如下所示：

```
1449550952: mosquitto version 1.4.5 (build date 2015-11-09 14:23:46-0800)
starting
1449550952: Using default config.
1449550952: Opening ipv4 listen socket on port 1883.
1449550952: Opening ipv6 listen socket on port 1883.
```

2 详细见：http://www.eclipse.org/mosquitto/download/。

MQTT 协议默认使用的是 1883 端口，所以它会监听 1883 端口。

接着，我们可以开始订阅一个名为 **device/2** 的主题，这个主题也可以是其他内容的字符串。而我们所做的只是定义好一个名称，来让订阅者可以获取想要的内容。

作为一个 MQTT 客户端，则需要分成两部分：发布者（pub）和订阅者（sub）。发布者（pub）需要发布消息给其他客户端。订阅者（sub）需要接受其订阅的发布者发布的信息。

运行下面的命令，我们就可以订阅 device/2 这个主题。

```
mosquitto_sub -v -t 'device/2'
```

这里的-v 用于输出发布的消息的详细信息，-t 则用于订阅相关的 MQTT 主题。

```
-v : print published messages verbosely.
 -t : mqtt topic to subscribe to. May be repeated multiple times.
```

然后，我们可以用 mosquitto_pub 来发布相应主题的信息：

```
mosquitto_pub -t 'device/2' -m 34
```

-m 参数用于要发送的消息的负载。

最后，我们就可以在运行订阅的页面看到接收到的相关信息。并且在我们运行 mosquitto server 的窗口会有相关的信息输出，如图 7-2 所示。

```
fdhuang@PHODAL > DesignIoT > master ✘ > mosquitto
1449552261: mosquitto version 1.4.5 (build date 2015-11-09 14:23:46-0800) starting
1449552261: Using default config.
1449552261: Opening ipv4 listen socket on port 1883.
1449552261: Opening ipv6 listen socket on port 1883.
1449552262: New connection from ::1 on port 1883.
1449552262: New client connected from ::1 as mosqsub/72421-PHODAL (c1, k60).
1449552263: New connection from ::1 on port 1883.
1449552263: New client connected from ::1 as mosqpub/72422-PHODAL (c1, k60).
1449552263: Client mosqpub/72422-PHODAL disconnected.
1449552264: New connection from ::1 on port 1883.
1449552264: New client connected from ::1 as mosqpub/72477-PHODAL (c1, k60).
1449552264: Client mosqpub/72477-PHODAL disconnected.
```

图 7-2　Mosquitto Server 启动过程

mosquitto 为我们提供了一些非常有用的功能，并且我们也可以用其附带的命令，来测试我们即将要编写的 MQTT 服务端代码。

7.1.2 创建 MQTT 服务

同我们在 HTTP 协议里所做的一样，我们也需要一个 MQTT 的库来加快我们的开发流程。这里将使用 MQTT.js——它是一个开源的 MQTT v3.1.1 的 Node.js 实现。

MQTT.js 除了是一个 MQTT 消息服务器，也提供了同 Mosquitto 一样的命令行工具。我们可以将这个框架安装到全局环境，如下：

```
npm install mqtt -g
```

安装完成后会提供相应的 pub 和 sub 命令，使用方式和 Mosquitto 类似。唯一不同的是，我们需要将 mosquitto_sub 拆成 mqtt sub，mosquitto_pub 拆成 mqtt pub，如下：

```
mqtt sub -v -t 'device/2'
mqtt pub -t 'device/2' -m 35
```

不过这里我们是要将其作为一个库来使用。根据 MQTT 的文档协议标准，其控制报文（Control Packet）可以分成 16 种形式，如表 7-1 所示。

表 7-1 MQTT 控制报文

名字	值	报文流动方向	描述
reserved	0	禁止	预留
connect	1	客户端到服务端	客户端请求连接服务端
connack	2	服务端到客户端	连接报文确认
publish	3	双向	发布消息
puback	4	双向	消息发布收到确认
pubrec	5	双向	消息发布已收到（确认交付的第一步）
pubrel	6	双向	消息发布释放（确认交付的第二步）
pubcomp	7	双向	消息发布完成（确认交付的第二步）
subscribe	8	客户端到服务端	客户端订阅请求
suback	9	服务端到客户端	订阅请求报文确认
unsubscribe	10	客户端到服务端	客户端取消订阅请求
unsuback	11	服务端到客户端	取消订阅报文确认
pingreq	12	客户端到服务端	心跳请求
pingresp	13	服务端到客户端	心跳响应
disconnect	14	客户端到服务端	客户端断开连接
reserved	15	禁止	预留

而在程序中，我们主要关注 connect、subscribe、publish、disconnect 这四个方法的实现，即关注在前面章节提到的获取和发布数据。

现在，可以将 mqtt.js 库添加到我们的 APP 中，如下：

```
npm install mqtt --save
```

随后，添加下面的代码到我们的 app.js 中：

```
var mqtt = require('mqtt');
var mqttServer = require('./mqttServer');

//...

mqtt.MqttServer(mqttServer).listen(1883, function () {
    console.log("mqtt server listening on port %d", 1883);
});
```

代码中的 mqttServer 文件则是我们用于处理 MQTT 消息的函数，其基本代码如下：

```
module.exports = function (client) {
    client.on('connect', function (packet) {

    });

    client.on('subscribe', function (packet) {

    });

    client.on('publish', function (packet) {

    });

    client.on('disconnect', function (packet) {

    });
};
```

在四个不同的事件中，我们对不同的事件进行处理，而主要的还是 subscribe 和

publish。在那之前我们可以先完成比较简单的 connect 和 disconnect 监听函数。在连接上服务端的时候，需要返回一个消息来告诉客户端你已经连接上了。这时我们需要 connack 的报文——服务端告知客户端，你已经连接成功了，这部分代码如下：

```
client.on('connect', function (packet) {
    self.clients[packet.clientId] = client;
    client.id = packet.clientId;
    console.log("CONNECT: client id: " + client.id);
    client.connack({returnCode: 0});
});
```

这时，我们向客户端返回的状态码为 0，表示连接已接受。根据 MQTT 协议，如表 7-2 所示是返回码与其状态描述。

表 7-2　MQTT 返回码与其状态描述

返回值	描述
0	连接已被服务端接受
1	服务端不支持客户端所请求的 MQTT 协议级别
2	客户端标识符是正确的 UTF-8 编码，但是服务端不允许使用
3	网络连接已经建立，但是 MQTT 服务不可用
4	用户名或密码错误
5	客户端未授权
6~255	预留

用我们在上面提到的 Mosquitto 命令，在连接服务端的时候，我们会得到类似于下面结果的一些日志：

```
CONNECT: client id: mosqsub/4893-PHODAL
```

这里的 **mosqsub/4893-PHODAL**，即我们的客户端 ID。如果你用的是 mqtt.js 的命令行，那么你得到的客户端 ID 可能是 **mqttjs_8e5f03f7**。这个 ID 可以让我们区分不同类型的客户端。

相应地，当我们断开与服务端的连接的时候，也会做一些相应的处理——记录断开的 ID 到日志，并断开客户端的数据流。

```
client.on('disconnect', function (packet) {
```

```
        console.log("DISCONNECT: client id: " + client.id);
        client.stream.end();
    });
```

完成上面的部分后，我们就可以开始对 subscribe 事件进行处理了。下面的代码是官方示例中的订阅事件的函数：

```
client.on('subscribe', function(packet) {
    var granted = [];

    console.log("SUBSCRIBE(%s): %j", client.id, packet);
    for (var i = 0; i < packet.subscriptions.length; i++) {
      var qos = packet.subscriptions[i].qos
        , topic = packet.subscriptions[i].topic
        , reg = new RegExp(topic.replace('+', '[^\/]+').replace('#', '.+')
+ '$');

      granted.push(qos);
      client.subscriptions.push(reg);
    }

    client.suback({messageId: packet.messageId, granted: granted});
  });
```

在这个函数里，我们会取出每一个订阅的 topic 和 qos，并且保存用正规表达式对主题进行一些特殊的处理——在发布的时候将用到这个表达式。在第 8 行代码中，先对 topic 中的+用[^\/]+进行替换，再替换 topic 中的#为.+。

这部分内容实际上是主题过滤器，+是用于单个主题层级匹配的通配符，如 sport/tennis/+可以匹配“**sport/tennis/player1**”和“**sport/tennis/player2**”，但是不能匹配“**sport/tennis/player1/ranking**”。而“#”则用于主题中任意层级的通配符的匹配，如 **sport/tennis/player1/#**，可以匹配 **sport/tennis/player1**”、“**sport/tennis/player1/ranking**”等。除此之外，在之前我们还会用“/”来划分每个级别，这和 URL 的写法是相似的。

接着，这些 rege 都会被保存到 client.subscriptions 数组中，并用 suback 方法告知

客户端它已经收到订阅。

在订阅的时候，我们还需要注意的是，MQTT 协议的三种不同的服务质量代表了三种不同的模式。MQTT 的三种不同服务质量（QoS）等级及描述如表 7-3 所示。

表 7-3　MQTT 三种服务质量等级及描述

QoS 级别	描述
QoS 0	最多分发一次
QoS 1	至少分发一次
QoS 2	仅分发一次

图 7-3 描述了这三种不同等级的 QoS，级别越高就意味着越复杂。

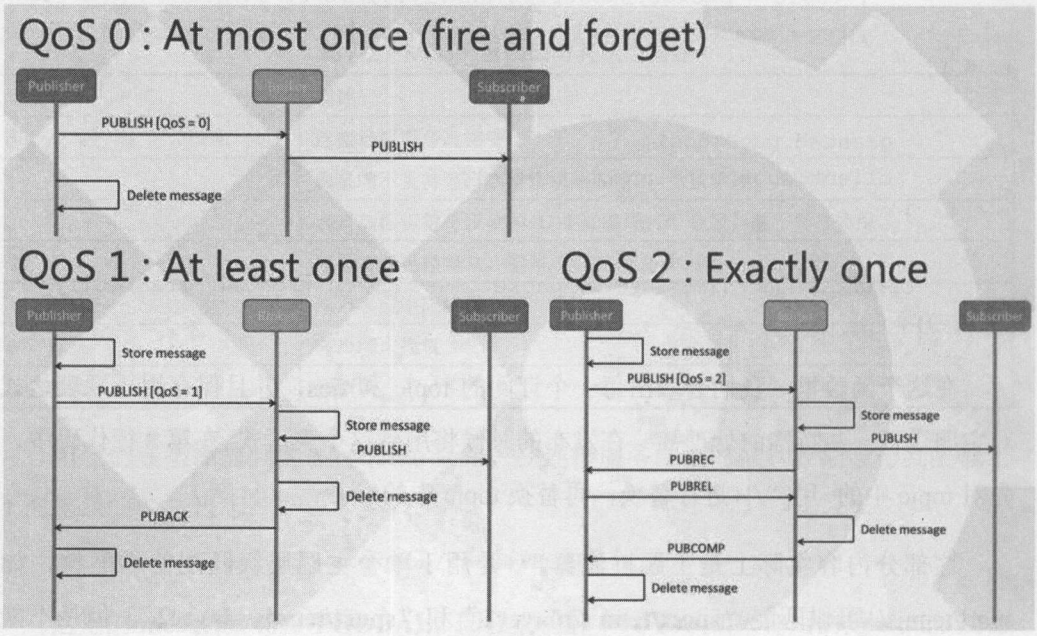


图 7-3　MQTT QoS

这里我们就不深入探索这部分内容了，读者可以参考官方相关文档及实现。

或许你注意到了，在我们订阅的时候，只是对客户端的请求进行了一些保存处理。真正用到它的地方是在我们的发布函数里。在这里，我们仍然使用 MQTT.js 的官方示例进行一些解释和说明。

```
    client.on('publish', function (packet) {
        console.log("PUBLISH(%s): %j", client.id, packet);
        for (var k in self.clients) {
            var _client = self.clients[k];

            for (var i = 0; i < _client.subscriptions.length; i++) {
                var subscription = _client.subscriptions[i];

                if (subscription.test(packet.topic)) {
                    _client.publish({topic: packet.topic, payload:
packet.payload});
                    break;
                }
            }
        }
    });
```

当我们发布一个新的消息的时候。我们将遍历所有现有的客户端（_client）连接，并取出每一个客户端（_client）中的 subscription。由于每个 subscription 都是一个主题过滤器，我们就可以用这个过滤器来验证当前发布的内容是否是客户端想要的主题。如果是客户端订阅的主题，就向这个客户端发布这条信息。

现在我们已经对 MQTT 的订阅机制有一些了解，下面让我们整合这个服务到我们的物联网系统中。

7.1.3 整合 MQTT 服务

现在，我们可以整合 MQTT 服务到原有的应用中了。由于连接和断开连接部分的代码和上节中的一样，这里就不一一描述了。

1. 添加订阅服务

首先，我们需要做的是在我们的订阅方法里添加一个简单的数据库查询。这可以保证当客户端第一次订阅的时候获得现有的值。

```
    client.on('subscribe', function (packet) {
        var payload = {user: 1, device: 1};
        db.findOrder(payload, 1, function (results) {
```

```
            var topic = packet.subscriptions[0].topic.toString();
            client.publish({
                topic: topic,
                payload: JSON.stringify(results)
            });
        });
    });
```

上面的代码和我们之前的 HTTP 协议中的查找传感器数据的代码基本上是一样的。如下是之前的 HTTP 协议中查找相关设备值的代码：

```
    var payload = {user: parseInt(req.params.user_id), device: parseInt
(req.params.device_id)};
    db.findOrder(payload, parseInt(req.params.result_id), function
(results) {
        return res.json(results);
    });
```

稍有区别的是最后对于数据的处理：在 HTTP 协议中我们直接返回了字符串，但是在 MQTT 协议中返回的时候我们需要添加相应的主题。在这里我们的 topic 需要从 packet.subscriptions 中取出来，如下是订阅时的数据包的数据：

```
  SUBSCRIBE(mosqsub/27575-PHODAL):
{"cmd":"subscribe","retain":false,"qos":1,
"dup":false,"length":13,"topic":null,"payload":null,"subscriptions":[{"topic":
"device/2","qos":0}],"messageId":9}
```

我们的 topic 是 subscriptions 数组中的第一个，我们可以从这里面拿到 topic，即上面代码中的 **packet.subscriptions[0].topic.toString()**。只拿第一个 topic 也就意味着，这里只支持一次订阅一个 topic。在完成一个简单的数据查询之后，我们可以解析相应的主题来获取相关的用户和设备数据。

在之前的实现里，我们已经有了一个用户名的选项。但是这个选项是用户 ID，在这里我们将用用户 ID 来充当用户名。对于服务端来说，用户和设备身份认证是一个很重要的话题。尽管不会在这个章节里详细说明，但是还是应该好好考虑这个问题。

MQTT 里的 CONNECT 报文包含用户名和密码字段，现在我们只有一个用户名

的选项，而我们的设备名则可以作为主题名。根据 mosquitto_pub 的帮助选项，我们可以发现这里面有一个可选参数为**-u**：

```
-u : provide a username (requires MQTT 3.1 broker)
```

这个参数的用处就是提供一个用户名。接着，我们可以升级一下发布脚本：

```
mosquitto_pub -t 'device/2' -m '{"temperature":3}' -u 1
```

然后，我们就可以在 connect 监听事件里判断一下是否包含用户名：

```
var user = null;
client.on('connect', function (packet) {
    self.clients[packet.clientId] = client;
    client.id = packet.clientId;
    console.log("CONNECT: client id: " + client.id);
    if (packet.username === undefined) {
        client.connack({returnCode: 4});
    }
    user = packet.username;
    client.subscriptions = [];
    client.connack({returnCode: 0});
});
```

当 packet.username 等于 undefined 的时候，会返回状态码 4——用户名或密码错误。如果包含用户名，那么就将用户名的值赋给 user 变量。在 subscribe 和 publish 的时候，我们可以使用这个变量来修改用户的设备值。

在取得用户名之后，我们需要对用户订阅的 topic 进行解析，从中取得设备的 ID。

```
var topicRegex = /device\/(\d)/;
if(!topicRegex.test(topic) || topicRegex.exec(topic).length < 1){
    return client.connack({returnCode: 6});
}
```

在上面的代码中，我们使用了正规表达式来获取设备的 ID。正则表达式是被用来匹配字符串中的字符组合的模式。在处理字符串的时候，使用正则表达式是一种非常有效并且“易懂”的方案。易懂只是对于看得懂的人来说的，对于不懂的人来

说就是天文，因此建议读者查看相应的书籍或者资料。

我们的正则表达式是 device/()，在这个表达式的两边的/是 JavaScript 用来标识这是一个正则表达式的。首先，我们的表达式是以 device 字母开头的，这表明它要匹配的字符串必须是 device。紧随其后的/是用于匹配符号/的，因为这个符号是特殊符号，所以需要有前面的“”表示转义。接着是()，其中的“()”用来分组——匹配输入字符串中重复的子表达式。由于我们只需要拿到其中的设备 ID，所以我们需要这个分组来划分其中的子表达式，即表达式中的 “,” 用来匹配数字。

当我们用上面的正则表达式去匹配 device/1 时，会得到一个数组['device/1', '1']。因为正则表达式会有一个默认的分组即整个表达式匹配的值，然后子表达式()则会变成数组中的第二个值，我们只需要取出分组中的值即可。

最后我们可以整合在上一节中提到的保存客户端订阅的代码。当用户订阅的时候，我们就可以告知用户之前的值，并在以后的更新中通知用户。在这里，我们所做的只是简单的复制、粘贴：

```
client.on('subscribe', function (packet) {
    var topic = packet.subscriptions[0].topic.toString();
    if (!/device\/(\d)/.test(topic) || /device\/(\d)/.exec(topic).length
< 1) {
        return client.connack({returnCode: 6});
    }
    console.log("SUBSCRIBE(%s): %j", client.id, packet);
    var deviceId = parseInt(/device\/(\d)/.exec(topic)[1]);
    var payload = {user: parseInt(user), device: deviceId};
    var granted = [];
    for (var i = 0; i < packet.subscriptions.length; i++) {
        var qos = packet.subscriptions[i].qos
          , topic = packet.subscriptions[i].topic
          , reg = new RegExp(topic.replace('+', '[^\/]+').replace('#',
'.+') + '$');

        granted.push(qos);
        client.subscriptions.push(reg);
        console.log(reg);
```

```
    }

    client.suback({messageId: packet.messageId, granted: granted});

    db.subscribe(payload, function (results) {
        client.publish({
            topic: topic,
            payload: JSON.stringify(results)
        });
    });
});
```

紧接着，我们就可以做发布服务了。

2. 添加发布服务

有了订阅服务之后，要添加发布服务变得很容易。同样我们也需要去处理主题是否匹配，如果不匹配就返回错误信息：

```
var topic = packet.topic.toString();
var topicRegex = /device\/(\d)/;
if (!topicRegex.test(topic) || topicRegex.exec(topic).length < 1) {
    return client.connack({returnCode: 6});
}
console.log("PUBLISH(%s): %j", packet.clientId, packet);
var deviceId = parseInt(topicRegex.exec(topic)[1]);
```

然后，还要将这些数据插入数据库中：

```
var payload;
try {
    payload = JSON.parse(packet.payload);
} catch (err) {
    console.log(err);
    return client.connack({returnCode: 6});
}
payload.user = parseInt(user);
payload.device = deviceId;
db.insert(payload);
```

由于我们无法确定传过来的数据是否是JSON格式的，我们需要做一个简单的try

catch 处理。如果不是 JSON 格式就返回一个错误码，如果是就解析这个数据，并向数据中添加 user id 和 device id，然后插入数据库中。

最后，修改上节的发布代码中的 payload 为这里的 payload：

```
for (var k in self.clients) {
    var _client = self.clients[k];

    for (var i = 0; i < _client.subscriptions.length; i++) {
        var subscription = _client.subscriptions[i];

        if (subscription.test(packet.topic)) {
            _client.publish({topic: packet.topic, payload: payload});
            break;
        }
    }
}
```

这样就可以向订阅者发布订阅信息了。

现在让我们来做一个简单的测试，启动服务器——在图 7-4 左下窗口中。在图 7-4 左上的窗口中，我们创建一个用户，ID 为 1，订阅 device/1 和 device/2 两个主题：

```
mqtt sub -v -t 'device/2' -t 'device/1' -u 1
```

在右上窗口中，我们只订阅用户 ID 为 1 的用户，并且主题为 device/2：

```
mqtt sub -v -t 'device/2' -u 1
```

现在，我们可以在发布窗口——图 7-4 右下，发布两条不同的信息：一个是 device/2，另一个为 device/1 的消息。当我们在 device/2 主题发布消息时，两个窗口应该都收到消息，当我们在主题 device/1 发布消息时，只有同时订阅两个窗口才会收到消息。发布消息代码如下所示：

```
mosquitto_pub -t 'device/2' -m '{"temperature":3}' -u 1
mosquitto_pub -t 'device/1' -m '{"temperature":3}' -u 1
```

最后的效果如图 7-4 所示。

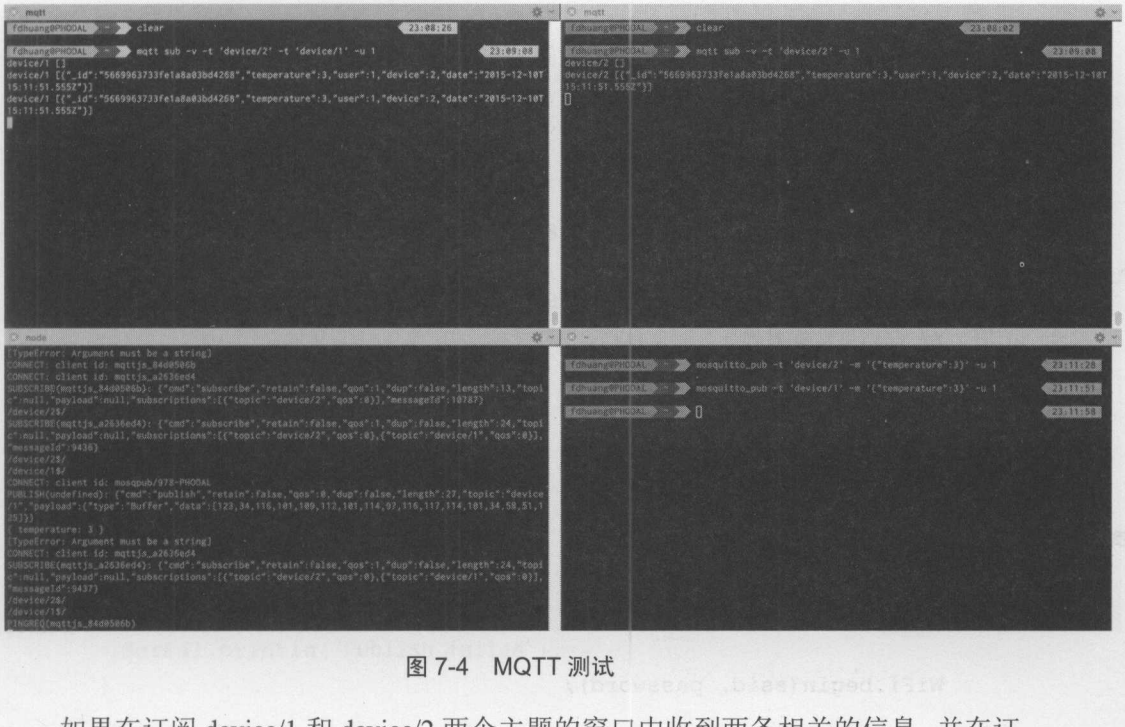


图 7-4　MQTT 测试

如果在订阅 device/1 和 device/2 两个主题的窗口中收到两条相关的信息，并在订阅主题 device/2 的窗口中只收到一条消息，说明我们的逻辑是正确的！

3. 整合硬件

现在，我们可以接上控制器来上传数据了。在这里我们将使用 NodeMCU 与 Arduino 开发环境作为示例来讲述这个过程。

我们需要一个名为 PubSubClient 的库，这是一个 MQTT 的客户端库。它支持 Arduino YUN 、Arduino Ethernet、Arduino WiFi、Intel Galileo、ESP8266，但是当前不能运行在 ENC28J60 芯片上。同之前的库安装方法一样，可以直接使用 Arduino IDE 安装这个库。稍微修改官方的示例代码，我们就可以得到下面的代码：

```
#include <PubSubClient.h>
#include <ESP8266WiFi.h>

const char* ssid = "...";
const char* password = "...";
```

```
char* topic = "device/2";  //订阅主题
char* server = "192.168.1.31"; //服务器 IP 或者网址


WiFiClient wifiClient;
PubSubClient client(server, 1883, callback, wifiClient);

void callback(char* topic, byte* payload, unsigned int length) {
  // 在这里处理消息
}

void setup() {
  Serial.begin(115200);
  delay(10);

  Serial.print("Connecting to ");
  Serial.println(ssid);

  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }
  Serial.println("");
  Serial.println("WiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());

  Serial.print("Connecting to ");
  Serial.print(server);

  if (client.connect("ESP8266Client", "1", " ")) {
    Serial.println("Connected to MQTT broker");
    Serial.print("Topic is: ");
    Serial.println(topic);
  }
  else {
    Serial.println("MQTT connect failed");
```

```
    Serial.println("Will reset and try again...");
    abort();
  }
}

void loop() {
  static int counter = 0;

  String payload = "{\"temperature\":3}";

  if (client.connected()){
    Serial.print("Sending payload: ");
    Serial.println(payload);

    if (client.publish(topic, (char*) payload.c_str())) {
      Serial.println("Publish ok");
    }
    else {
      Serial.println("Publish failed");
    }
  }
  ++counter;
  delay(5000);
}
```

代码中的第 4~8 行定义了要连接的 WiFi、要发布的主题、MQTT 服务器的网址。接着，我们创建了一个 WiFi 客户端 wifiClient 和 MQTT 客户端 client。当服务端向我们发出信息的时候，我们就需要去处理，这部分的处理逻辑由 callback 函数来处理。因为当前没有相关的消息需要处理，这个函数是空的。随后在 setup()函数里，我们尝试去连接网络，直到下面的 if 语句：

```
if (client.connect("ESP8266Client", "1", " ")) {
  Serial.println("Connected to MQTT broker");
  Serial.print("Topic is: ");
  Serial.println(topic);
}
```

连接我们的服务端，下面是 PubSubClient 中的 connect 的重载函数——相同的函

数名，但是参数列表不相同。在使用的时候，我们只需要传相应的参数进去即可。编译器将通过我们传入的参数的类型来调用相应的方法，在这里我们调用了下面的第二个函数。我们定义了客户端 ID 是 ESP8266Client，用户名是 1，密码为空。

```
boolean connect(const char* id);
boolean connect(const char* id, const char* user, const char* pass);
boolean connect(const char* id, const char* willTopic, uint8_t willQos,
boolean willRetain, const char* willMessage);
boolean connect(const char* id, const char* user, const char* pass, const
char* willTopic, uint8_t willQos, boolean willRetain, const char*
willMessage);
```

连接成功后，我们就循环执行 loop 函数中的语句。不断地发布数据 **{"temperature":3}**，我们只需要接上真实的传感器就可以用 MQTT 协议实现实时数据更新。最后，运行结果如图 7-5 所示。

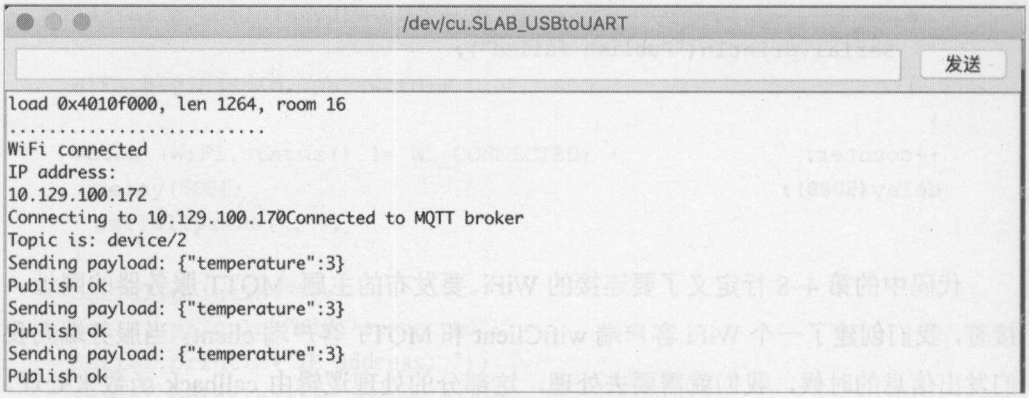


图 7-5　MQTT ESP8266 连接

在服务端则会收到类似于下面的信息：

```
CONNECT: client id: ESP8266Client
PUBLISH(undefined): {"cmd":"publish","retain":false,"qos":0,"dup":false,
"length":27,"topic":"device/2","payload":{"type":"Buffer","data":[123,34,116,1
01,109,112,101,114,97,116,117,114,101,34,58,51,125]}}
```

最后，再查看数据库，会发现数据已经上传到里面了——我们已经成功地集成 MQTT 服务了。

7.1.4 MQTT-SN

在近几年里，由于无线传感器网络（Wireless Sensor Networks）越来越受关注，从商业的角度来说，它成本很低，从技术的角度来说，它很容易部署。在这些嵌入式系统中使用 MQTT 协议会花费更多的系统资源——MQTT 需要一个网络基础条件，如 TCP/IP 协议支持。这时就产生了 MQTT For Sensor Networks，即用于传感器网络的 MQTT，它是一个面向无线传感器网络的订阅/发布协议。并且 MQTT 也为低功耗、资源受限的嵌入式设备做了优化。

尽管 MQTT-SN 在设计上尽可能与 MQTT 保持一致，但是它们在实现上也有一些差异。如表 7-4 所示是两个协议之间的一些对比。

表 7-4 MQTT 与 MQTT-SN 对比

比较类型	MQTT	MQTT-SN
传输类型	可靠点对点流模式	不可靠的数据报
通信方式	TCP/IP	Non-IP 或 UDP
网络传输	Ethernet、WiFi、3G	ZigBee、Bluetooth、RF
最小消息	2 个字节	2 个字节
最大消息	≤24MB	< 128 个字节
电池供电	X	√
休眠支持	X	√
QoS -1（哑客户端）	X	√
主题标识符	X	√
网关自动发现和回退	X	√

由于 MQTT-SN 协议在网络传输上的一些限制，在这里我们仅对 MQTT-SN 做一个简单的介绍，不做深入的研究。

MQTT 协议与 HTTP 协议都依赖于底层的 TCP/IP 协议，而实现这个协议对于嵌入式设备来说都是一个负担。这时，我们就需要 CoAP 协议。

7.2 CoAP

COAP 源自 Constrained Application Protocol，即受约束的应用协议。它是一种软

件协议，旨在以非常简单的方式让电子设备能够在互联网上进行交互式通信。它专门针对小型低功率传感器、开关、阀门，以及需要被控制或被远程监督的设备——标准的 Internet 网络类似的组件。CoAP 被设计为很容易转换为 HTTP 与 Web 简化集成，同时也能满足特殊的要求，例如多播支持、非常低的开销和简单性。多播、低开销及简单性是 Internet 极其重要物联网（IOT）和机器对机器（M2M）设备。在这些设备里有太多的内存和电源，比传统的互联网设备多得多。因此，效率对于它们来说是非常重要的。

简单来说，CoAP 简化了 HTTP 协议，并仅仅提供了 REST 的四个方法：PUT、GET、POST 和 DELETE。并且 CoAP 用更紧凑的二进制头替换 HTTPU 中使用的文本头。

CoAP 协议的交互模型类似于 HTTP 协议中的 C/S（客户端/服务端）模型。一个客户端发现的 CoAP 请求等价于 HTTP 协议的请求——都是由客户端发给服务端，并且请求操作（使用请求方法码）某个资源。接着，服务器发出一个响应及其响应码，这个响应可能会包含一个资源的表示。而不同的是：CoAP 协议处理这些异步的交互是在面向数据报的运输协议上，如 UDP。如图 7-6 所示是 CoAP 协议中的 CoAP 协议的抽象层级结构。

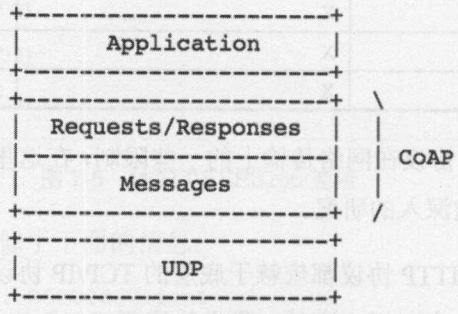


图 7-6　CoAP 抽象层级结构

CoAP 使用 UDP 的好处在于，对于资源受限的通信的 IP 网络，HTTP 不是一种可行的选择，它占用了太多的资源和太多的带宽，而对于物联网这种嵌入式设备来说，这是我们需要优先考虑的问题。

同 MQTT 协议的 QoS 类似，CoAP 也定义了四种类型的消息，分别如下。

- Confirmable（CON）：需要收到确认的消息。
- Non-confirmable（NON）：不需要收到确认的消息。
- Acknowledgment（ACK）：表示确认一个 Confirmable 类型的消息已收到。
- Reset（RST）：表示一个 Confirmable 类型的消息已收到，但是不能处理。

7.2.1 CoAP 协议示例

同样，我们也需要一个相应的 CoAP 协议的工具软件来调试代码，这里先介绍一下 libcoap。libcoap 是 CoAP 协议的 C 语言实现，它和 mosquitto 类似，也提供了 server 和 client 功能。

libcoap 的安装方法和我们之前举例的方法一样。安装完 libcoap 后，libcoap 会提供两个工具：

- coap-client，CoAP 客户端。
- coap-server，CoAP 服务端。

先在命令行上运行 coap-server，来启动一个 CoAP 服务器。接着，在另外一个命令行上运行：

```
coap-client -m get coap://localhost/
```

就会得到如下所示的结果：

```
v:1 t:0 tkl:0 c:1 id:59515
This is a test server made with libcoap (see http://libcoap.sf.net)
Copyright (C) 2010--2013 Olaf Bergmann <bergmann@tzi.org>
```

这表明，我们已经连接上这个 CoAP 服务器了。

或许你还注意到了上面的 **v:1 t:0 tkl:0 c:1 id: 59515** 一行，这一行很特别——都是一些缩写，它们是 CoAP 的 header。这些 header 都有其特殊含义，如表 7-5 所示。

表 7-5 CoAP 的 header

缩写	全称	类型	用途
v	版本（version）	两位无符号整数	表明 CoAP 的版本号（当前这个值必须设置为 1）
t	类似（type）	两位无符号整数	表明这个消息的类型是 CON（需要确认）或者 NON（不需要确认）
tkl	令牌长度（Token Length）	4 位无符号整数	表示可变长度令牌字段的长度（0~8 字节）
c	状态码（code）	8 位无符号整数	表示这个请求的状态：请求（0）、请求成功（2）、客户端错误响应（4）、服务器错误反应（5）。特殊情况下，0.00 表示空消息
id	消息 ID（message ID）	网络字节顺序的 16 位无符号整数	用于检测消息是否重复

除了 licoap，还有几个不错的工具可以使用：

- CoAP-cli，一个基于 NodeJS 的 CoAP 命令行工具，其核心基于 Node-CoAP 库。
- FireFox Copper，一个 Firefox 的图形用户界面插件。

Node.js 的 CoAP CLI 安装命令如下：

```
npm install coap-cli -g
```

在 coap-cli 中，一共有 4 个方法，分别表示 REST 的 4 种不同的方式:

```
Commands:

get                     performs a GET request
put                     performs a PUT request
post                    performs a POST request
delete                  performs a DELETE request
```

在这里，我们用 coap://vs0.inf.ethz.ch/来做一个简单的测试：

```
coap get coap://vs0.inf.ethz.ch/
(2.05) ************************************************************
CoA
```

上面的 2.05 是一个状态码，相当于 HTTP 协议中的 200 OK，即请求已成功。CoAP 请求过程如图 7-7 所示。

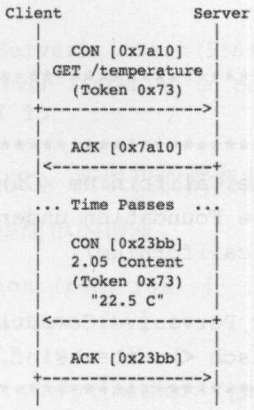


图 7-7　CoAP 消息请求过程

除了上面的两个工具，我们也可以在 Firefox 浏览器上使用 Firefox Copper 来访问。Copper 是 Firefox 浏览器上的一个 CoAP 协议的用户代理插件。通过这个插件开发人员可以直接在浏览器上访问物联网，并且可以直接与服务端交互——获取、上传、删除等。

我们只需要在浏览器的地址栏输入相应的 URI，单击 GET 按钮，就可以获取这个资源的数据，其截图如图 7-8 所示。

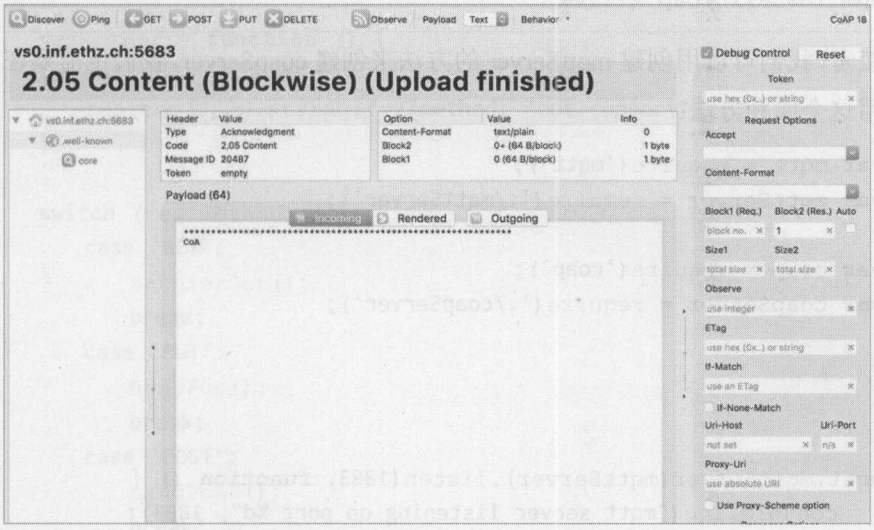


图 7-8　Firefox Copper

它将读取到下面的内容：

```
************************************************************
CoAP RFC 7252                              Cf 1.0.0-SNAPSHOT
************************************************************
This server is using the Californium (Cf) CoAP framework
published by the Eclipse Foundation under EPL+EDL:
http://www.eclipse.org/californium/

(c) 2014, Institute for Pervasive Computing, ETH Zurich
Contact: Matthias Kovatsch <kovatsch@inf.ethz.ch>
************************************************************
```

或者你注意到了，上面的返回结果中包含了一个名为 Californium (Cf) 的 CoAP 框架。它是 Eclipse 基金会推出的 CoAP 协议的 Java 语言实现。如果你想在服务端使用 Java 语言，就可以使用这个框架来完成开发。

7.2.2 创建 CoAP 服务

整合 CoAP 服务器的过程和整合 CoAP 的步骤类似，但是代码则和 HTTP 协议的过程差不多。首先，我们需要安装 Node.js 的 coap 库，代码如下：

```
npm install coap --save
```

接着，我们可以用创建 mqttServer 的方法来创建 coapServer。我们也需要在 **app.js** 中引用这个库和方法：

```
var mqtt = require('mqtt');
var mqttServer = require('./mqttServer');

var coap = require('coap');
var coapServer = require('./coapServer');

//...


mqtt.MqttServer(mqttServer).listen(1883, function () {
    console.log("mqtt server listening on port %d", 1883);
});
```

```
coap.createServer(coapServer).listen(5683, function () {
    console.log("coap server listening on port %d", 5683);
});
```

CoAP 协议默认的端口是 5683，所以我们需要监听 5683 端口。然后，可以创建一个基本的处理函数，处理一些简单的请求：

```
module.exports = function (req, res) {
    var handlerGet = function () {
        res.code = '2.05';
        res.end(JSON.stringify({method: 'get'}));
    };

    var handPut = function () {
        res.code = '2.01';
        res.end(JSON.stringify({method: 'put'}));
    };

    var handPost = function () {
        res.code = '2.01';
        res.end(JSON.stringify({method: 'post'}));
    };

    var other = function () {
        res.code = '4.00';
        res.end(JSON.stringify({method: 'not support'}));
    };

    switch (req.method) {
        case 'GET':
            handlerGet();
            break;
        case 'PUT':
            handPut();
            break;
        case 'POST':
            handPost();
            break;
        default:
```

```
            return other();
        }
    };
```

由于 CoAP 库的限制，这里的函数可能并没有上面的函数易懂。但是主要还是 switch 在起作用，我们会判断请求的方法，即 req.method，依据不同的请求方法来调用不用的处理函数。当我们需要 GET 数据的时候，就会调用 handleGet()方法。这个方法会返回状态码 2.05，并用 res.end 返回字符串{"method": "get"}，同时结束响应。如果我们请求的方法不在 switch 语句的列表里，就会转由 other 方法来处理。

如表 7-6 所示是一些我们将会用到的状态码，更详细的状态码请参见 CoAP 协议的文档。

表 7-6　CoAP 的状态码

状态码	表述	说明
2.01	Created	资源已创建，仅用于 POST 和 PUT 请求
2.02	Deleted	资源已删除，仅用于 DELETE 请求
2.05	Content	类似于 HTTP 协议中的 200 "OK"，仅用于 GET 请求
4.00	Bad Request	类似于 HTTP 协议中的 400 "Bad Request"，即请求无效
4.01	Unauthorized	未授权
4.03	Forbidden	禁止
4.04	Not Found	未找到
4.05	Method Not Allowed	方法不允许
5.01	Not Implemented	未实现

现在，我们可以运行代码 **node app.js**，然后用上文中提到的 CoAP 的测试工具来测试我们的代码：

```
coap-client -m get 'coap://127.0.0.1:5683/'
```

运行上面的代码，应该会得到类似（除了 id 不一样）下面的结果：

```
v:1 t:0 tkl:0 c:1 id:45268
{"method":"get"}
```

我们已经可以运行一个基本的 CoAP 协议的服务了，下面我们整合这个服务到应

用中。

7.2.3 整合 CoAP 服务

在开始之前，先让我们来看看最后需要的请求的 URI。URI 相当于我们的一个 API，这里是程序的入口。

```
coap://127.0.0.1:5683/result?user=1&device=1
```

这里的 result 相当于 PATH，也称为 URI-PATH。“?”后面的内容即要查询的参数，我们查询了 user=1 并且 device=1 时的结果。尽管这里的格式并不符合 RESTful 的习惯，但是我们还是简单地说明一下这种格式。

URI-Path 和 URI-Query 都会存储在请求的 options 变量中，可以遍历并取出这个值。

```
var uriQuery = {};
var existBlock = false;
for (var i = 0; i < req.options.length; i++) {
      if (req.options[i].name === 'Uri-Query') {
          var query = req.options[i].value.toString().split('=');
          uriQuery[query[0]] = parseInt(query[1]);
          existBlock = true;
      }
   }
```

首先，我们定义了一个 uriQuery 的对象，用来存储生成的 query，以及 existBlock，用于判断是否存在 query 这样的结果。接着在 for 语句里，遍历每一个 Option，如果这个 Option 的键（name）是 Uri-Query，那么我们就取出这个值，如 user=1 是一个，device=1 则是另外一个。并用 toString()将其转换为字符串，接着用 split 通过 = 分割符将字符串分割成字符串数组。最后将数组以 Key 和 Value 的形式存储，结果就会变成**{"user":1, "device": 1"}**。

为了更 RESTful，用一个比较丑陋的方法替换上面的方法：

```
var urlArray = req.url.split("/");
for (var i = 1; i < urlArray.length; i = i + 2) {
```

```
        uriQuery[urlArray[i]] = parseInt(urlArray[i + 1]);
    }
```

我们将请求的 URL（即/user/1/device/1）以 / 分隔符分隔，并以两个为一组，以第一个为键，第二个为值。虽然也会达到同样的效果，但是我们需要在这里添加更多的错误处理。

但是无论哪一种方法，都不会影响接下来的处理。

当我们在客户端发出一个 GET 请求的时候，这个请求就会走到 handleGet 方法中。在这个方法里我们只需要去查询数据库中的相应结果，并返回状态码 2.05。代码如下所示：

```
var handlerGet = function () {
    var payload = {user: uriQuery.user, device: uriQuery.device};
    db.find(payload, function (dbResult) {
        res.code = '2.05';
        res.end(JSON.stringify({result: dbResult}));
    });
};
```

其主要的处理逻辑和 HTTP 协议并没有太大区别。

对于 POST 和 PUT 请求也是类似的，不过在我们开始处理之前，需要判断一个用户的 ID 和设备的 ID 是不是有效的。

```
    var userId = parseInt(uriQuery.user);
    var deviceId = parseInt(uriQuery.device);

    if (isNaN(userId) || isNaN(deviceId)) {
        res.code = '4.01';
        return res.end(JSON.stringify({"error": "username or device
undefined"}));
    }
```

parseInt 方法将解析用户 ID 和设备 ID，如果它们返回的是 NaN，那么将会返回一个 4.01 状态码，并返回**{"error": "username or device undefined"}**消息。

随后，我们可以试着去解析用户要创建的数据。

```
var data;
try {
    data = JSON.parse(req.payload.toString());
} catch (err) {
    res.code = '4.04';
    res.end(err);
}
data.user = userId;
data.device = deviceId;
```

如果解析不成功，将会返回状态码 4.04，并告诉用户错误的相关信息。如果可以成功地解析数据，则会查找是否有已经存在的数据。有，则会更新这个数据，没有，则会插入一个新的数据。

```
db.find(payload, function (results) {
    if (results.length > 0) {
        db.update(data);
        res.code = '2.01';
        res.end(JSON.stringify({method: 'update'}));
    } else {
        db.insert(data);
        res.code = '2.01';
        res.end(JSON.stringify({method: 'insert'}));
    }
});
```

现在，我们已经能够处理 GET、POST、PUT 请求！它处理数据的方式和 HTTP 协议是如此相像，我们甚至不需要在代码上做太多的修改。只是改一下处理数据的方式，以及修改一些文档，就可以完成对这个协议的支持。

启动 Server，让我们来做一个简单的 POST 数据的测试：

```
coap-client -e '{"temperature": 3}' -m put
'coap://127.0.0.1:5683/result?user= 1&device=11'
```

然后再请求这个数据：

```
coap-client -m get 'coap://127.0.0.1:5683/result?user=1&device=11'
```

最后，我们就会成功地得到下面的结果：

```
v:1 t:0 tkl:0 c:1 id:26647
{"result":[{"_id":"566ed52390602ffaf9948852","temperature":3,"user":1,"dev
ice":11,"date":"2015-12-14T14:41:39.507Z"}]}
```

当我们为系统添加了 CoAP 协议支持，系统就已经能够支持主流的物联网协议了。

7.3 小结

本章介绍了 MQTT 协议、CoAP 协议，及其一些基本概念，并将这两个协议整合到服务层中，让服务层支持了更多的协议。当然物联网不仅仅只有 HTTP、MQTT、CoAP 协议，不同的厂商出于利益需要都会推广自己的相关协议，但是它们之间是相通的。我们所需要做的就是为不同的协议添加适配器，正如在第 3 章中提到的物联网架构。

我们在应用程序上添加不同的适配层来添加对不同协议的支持。我们构建了一个以数据库为核心的应用程序层，不同的协议可以在这个基础上进行通信。

然而，这其中有一些问题需要注意，尽管我们使用同一个方法来处理数据，但是处理请求用户信息和设备信息都在某个协议中实现，当我们需要添加一些额外的支持时，如基本的密码和用户名验证、Token 等，需要在不同的协议上实现一遍。这时，需要将这些逻辑交由应用程序来解决。

7.4 相关阅读资料

[1] [美] Goyvaerts Jan，Levithan Steven．正则表达式经典实例[M]．郭耀译．北京：人民邮电出版社，2010．

第8章

智能与安全

本章内容

- 总结本书的物联网系统的用途
- 智能化物联网的基本知识
- 物联网安全的基本介绍

在前面几章里，介绍了如何去搭建一个物联网系统。我们花费了大量的时间在创建物联网应用，连接硬件到我们的物联网。本章将关注于一些额外的话题，如安全、智能、私有化。

当我们把各式各样的设备连接上网络之后，会发现一个非常尴尬的问题——我们还需要手动去控制这些设备。对于工业应用来说，开发者可能只需要能收集这些设备的数据，并将这些数据可视化出来即可。而对于家庭自动化及智能家居来说，这只是一个开始。在智能家居这个领域，智能是一个很有意思的话题。越来越多的人将精力投入到了这个领域，生产出了一些定时开关、定时插座、定时摄像头等定时设备，接着再冠以智能的头衔。它不能够自己学习、无法识别特定的模式，它真的能称为智能吗？

出于安全考虑，办公室、工厂等环境可能会采用一个自治的物联网子网，并有限制地与全球的网络互联。而在一些更特殊的环境中，它们甚至不与网络相连，只

会采用私有化的内部网络。除此以外，在一些更需要关注安全的话题里，人们会考虑使用自己的传输介质，使用自己特有的网络传输协议。制定并实现一个网络协议不是一件复杂的事，只需要在自己的网络中实现即可。但是在与外部网络互联的时候，还是需要与外部的网络使用同样的协议。

8.1 回顾我们的物联网系统

我们在第 3 章中提到了一个理想的物联网架构，如图 8-1 所示。

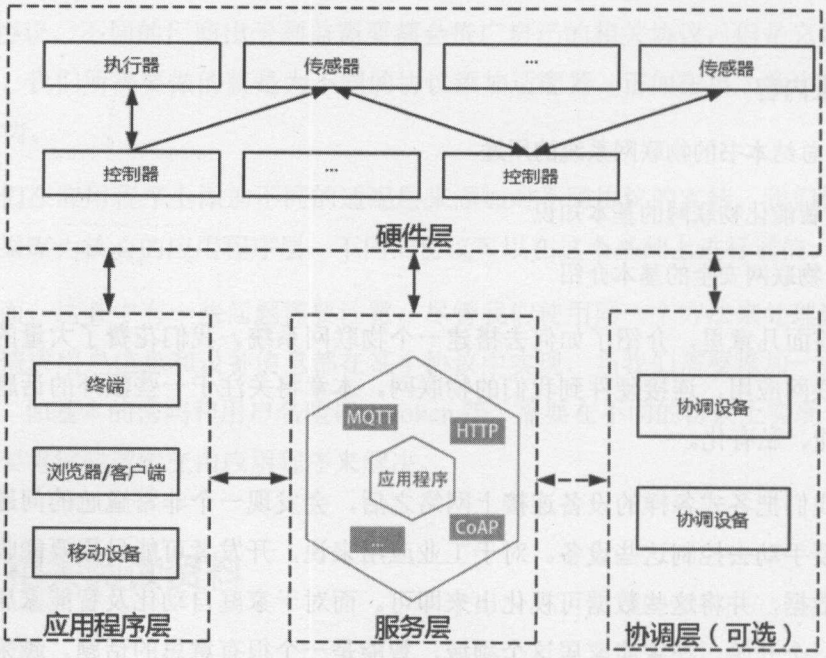


图 8-1 理想的物联网架构

并且在随后的章节里，我们实现了一个类似的物联网架构。我们拥有：

- 网页端的控制台。
- 跨平台的手机 APP。
- 简单实用的仪表盘。

- 支持 HTTP、CoAP、MQTT 协议。
- 使用的是分布式文件存储的数据库 MongoDB。
- ……

将我们学习到的知识点放到上面的架构图中，就变成了图 8-2。

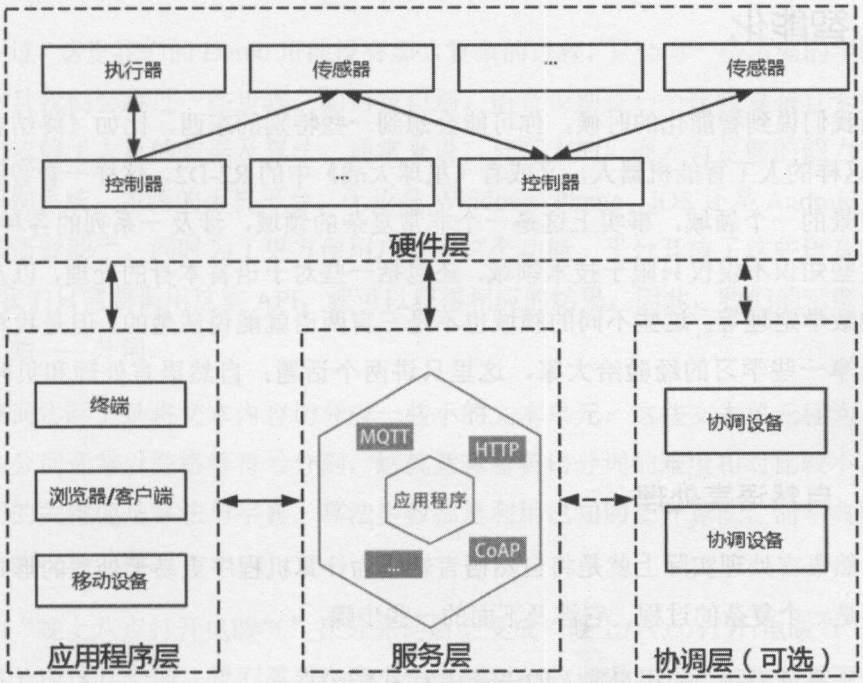


图 8-2　本书的物联网架构

我们使用 Raspberry Pi 作为协调设备来连接 Arduino 和服务端，由 Arduino 来获取传感器数据、控制 LED。我们还开发了一个 APP 和网页端应用，我们可以在手机和网页上控制这个 LED。并且还可以在手机和网页端查看传感器数据的值。同时，在我们的服务层上支持 HTTP、CoAP、MQTT 协议。

尽管在我们的示例中，使用了 Raspberry Pi 作为协议设备，但是我们也可以使用 NodeMCU 或者 Raspberry Pi 直接充当控制器和网络连接设备。

而这其中历尽了很多的坎坷——由于涉及了太多的知识点，也有很多的难点。

但是我们实现了这样一个系统也是让人很兴奋的事，我们只需要扩展这个系统，就可以将其运用到日常生活中。幸运的是，掌握了这些知识点就可以构成一个完整的系统。而那些在商业中使用的物联网系统也使用了相似的架构，在遇到相似的系统时，我们很容易理解其结构。并且，我们也可以构建出一个商业级的物联网系统了。

8.2 智能化

当我们说到智能化的时候，你可能会想到一些特别的东西，比如《终结者》中T-800这样的人工智能机器人，又或者《星球大战》中的R2-D2。这样一看智能化是很有前景的一个领域，事实上这是一个非常复杂的领域，涉及一系列的各种知识。并且这些知识不仅仅只限于技术领域，还包括一些对于语言本身的处理，以及一些复杂的数学定理等。这些不同的领域也不是三言两语就能说清楚的。但是我希望能在此分享一些学习的经验给大家，这里只讲两个话题：自然语言处理和贝叶斯分类器。

8.2.1 自然语言处理

自然语言处理实际上就是将自然语言转化为计算机程序更易于处理的形式。这个过程是一个复杂的过程，它涉及下面的一些步骤：

（1）语音识别。将用户输入的语音转为文本形式。

（2）中文自动分词。实则是将上一步生成的文本，切分成一个一个单独的词。

（3）词性标注。为上一步分成的词指派一个合适的词性，也就是要确定其是名词、动词、形容词或其他词性。

（4）句法分析。将句子中的词语语法功能进行分析，如“我写了本书”中的“我”是主语，“写”是谓语。

（5）上下文分析（可选）。在这一步我们会根据一些过往的经验来推测用户这时所要表达的意思，它依赖于收集的数据、上下文、领域相关知识等。

（6）感知并执行。这一步需要将用户的输入映射到当前的一些情景，根据当前的领域做一些特殊的处理，最后执行相关的一些操作。

在这个过程中会涉及很多不同领域的知识，处理起来也不是一朝一夕就能完成的。这就是为什么这个领域在讨论了几十年后，才开始逐渐展露头角。尽管它看上去很复杂，但是幸运的是我们可以做一个简单的程序来模拟这些步骤。

不过，这里我们的 Demo 可能没有那么复杂的过程，只会有一些浓缩的步骤及算法，并且我们会跳过一些步骤，如语音识别。语音识别是一个比较复杂且专业的话题，它依赖于大量的数据及算法。通常来说，只有大的厂商才有足够的能力去开发语音识别系统。主流的手机平台，无论是 Windows Phone、iOS 还是 Android，都会搭建其语音助手。同时为了更方便用户使用这个功能，平台开放了这些语音助手的 API。我们只需要调用这些 API，就可以获得相应的结果。因此，我们的示例将从第二步开始——分词。

分词实际上是将文本内容切分成一些小的文本单元，这些文本单元称为词条。英文的分词通常以空格等符号分割，这就意味着英语分词的难度相对比较小。而中文分词的关键便是算法与字典，算法多数都是利用已知的公开算法，而字典则是比较麻烦的收集过程。

如“晚上八点打开电暖气”在分完词后会变成“晚上/八点/打开/电暖气”。为了将内容分成这几个词，我们需要在字典里有相对应的词语。因此，我们需要先初始化词典：

```
var dict = ['晚上','八点','打开','电暖气'];
```

虽然这里的字典很小，但是它可以工作。接着，我们需要去寻找一个比较不错的词法分析算法。由于篇幅所限，这里不详细说明这些词法，读者可以自行参阅相关的代码。我们将使用 Trie 树——一种高效的字符串前缀处理结构，来进行分词的处理。创建一个这样的对象，接着初始化字典，然后分词，如下所示：

```
var _trie = new Geng.trie();
var sentenct = "晚上八点打开电暖气";
var dict = ['晚上','八点','打开','电暖气'];
```

```
_trie.init(dict);
var words = _trie.splitWords(sentences)
```

接着，我们可以为词法分析器添加一些规则。并进行一些分类，如“打开”和“开启”都返回“open”等。

```
var clock = 0;
var deviceId = 0;
var open = false;

var _lexer = new Geng.lexer();
  _lexer.addRule(/晚上/, function () {
    clock = clock + 12;
  });

  _lexer.addRule(/八点/, function () {
    clock = clock + 8;
  });

  _lexer.addRule(/开启|打开/, function () {
    open = true;
  });

  _lexer.addRule(/电暖气|电暖|暖气/, function () {
    deviceId = 1;
  });

words.forEach(function (word) {
    _lexer.setInput(word);
    _lexer.lex();
  });
```

在上面的代码里，我们添加了一些规则：

（1）当匹配到时间是晚上的时候，时间就会加上 12 小时。

（2）当时间是 8 点的时候，我们的时间又会加 8。

（3）当匹配到开启或者打开的时候，open 的值都会是 true。

（4）当匹配到电暖气时，这个设备就是 1。

接着，我们会遍历 words 这个数组，对其进行分析。在分析完上面的结果后，应该会有一个对应的值，即时间是 20 点、open 值为 true、设备 id 为 1。

最后，就是对分析的结果进行处理。我们先新建了一个名为 devices 的对象，对设备进行处理，在这里我们只是简单地打印了一些日志。

```
function devices(id) {
  this.deviceId = id;
}

devices.prototype.openAt = function (clock) {
  console.log("Device " + this.deviceId + " will open at" + clock);
};
```

而后，根据上一步的分析结果，我们会做一些相应的处理：

```
if (open) {
  var device = new devices(deviceId);
  device.openAt(clock);
}
```

浏览器的 Console 就会输出下面的一些内容：

```
Device 1 will open at 20
```

尽管这是一个非常简单的自然语言处理的例子，但是看上去非常接近我们想要的智能系统。和物联网系统一样，我们所要做的就是在上面添加更多的设备支持和处理，使其更接近日常的使用。

8.2.2 机器学习之贝叶斯分类器

在第 6 章中，我们使用了趋势图来展示温度的数据，这样的数据只能让我们人为地去预测温度的变化。当我们处理更多的数据时，就需要依赖算法来帮我们解决问题。例如我们收集到了很多传感器数据，我们没有保证这些传感器在所有的时刻都是正常工作的。有时，可能只是一辆汽车走过就会影响系统的值，我们需要使用一些过滤算法来去除这些偶然因素。在一些特殊的情况下，我们无法从原始数据中获得我们所需要的信息。如垃圾邮件的检测，我们无法从一个单一的文字来判断邮

件是否是垃圾邮件。而当它们出现一些相似的特征的时候，我们就可以判断这个邮件是垃圾邮件。

这时候就需要机器学习来帮助我们解决问题。机器学习的主要目的是设计和分析一些让计算机可以自动“学习”的算法。机器学习是一个很复杂的领域，涉及概率论、统计学、逼近论、凸分析、计算复杂性理论等多门学科。通过机器学习算法，我们可以从数据中自动分析获得规律，并对数据进行预测。接着，让我们来看一些机器学习的现实应用的例子。

（1）推荐系统。当你在亚马逊、京东或者当当上购买完某本技术书籍的时候，这些网站就会向你推荐一些相似或者相关的书籍。这些相关书籍的出现，有时只是因为读者 A 买了这本书，又买了一本名为 a 的书，读者 B 又做了相似的事，系统就认为这两本书之间存在着关联性。

（2）垃圾邮件过滤。在信息时代，每天我们都会收到大量的垃圾邮件。如果你很少使用邮件，也可以打开邮箱看看有多少垃圾邮件。当我们把一个邮件标记为垃圾邮件的时候，这个记录就会发送给服务器。如果有相当多的用户将发送者的邮件标记为垃圾邮件，那么这个发送者的邮箱就会被列入黑名单。而这只是简单的过滤，复杂的过滤则会直接由邮件的内容来判定。

对于物联网的海量数据来说，机器学习是非常有用、也很难的一门技术。下面让我们来看一个比较简单实用的机器学习推荐算法——朴素贝叶斯算法。它是一个概率分类算法，也是贝叶斯分类算法中应用最为广泛的分类算法之一。它以英国数学家贝叶斯发现的贝叶斯定理为基础，用于描述两个条件概率之间的关系。它的主要用途是通过已知的三个概率函数推出第四个，这就是为什么它特别适合于机器学习的原因。

作为一个依赖于特征来进行分类的算法，我们需要划分一些简单的特征，例如：好与坏，过去与现在。接着，需要录入一些数据才能真正使它工作。让我们来看一个由 Tolga Tezel 编写的 Node.js 的 bayes 库中的示例：

```
var bayes = require('bayes')
```

```
var classifier = bayes()

classifier.learn('amazing, awesome movie!! Yeah!! Oh boy.', 'positive')
classifier.learn('Sweet, this is incredibly, amazing, perfect, great!!',
'positive')

classifier.learn('terrible, shitty thing. Damn. Sucks!!', 'negative')
```

接下来如果我们执行：

```
classifier.categorize('awesome, cool, amazing!! Yay.')
```

将返回 **positive**，因为在上面的训练（执行 learn 方法）中我们输入了一些相关的特征，并对上面的单词进行分类。如果是中文的话，我们需要借助于上一节中的分词方法。

在这一节中，我们向读者展示了一些自然语言处理和机器学习的用法，更多的内容还需要读者自己去探索。

8.3 安全与隐私

在互联网的 Web 应用中，安全是一个非常重要的话题。从互联网到物联网有太多共有的元素，使用数据存储、拥有用户授权和验证、使用中央服务器等。在多数的攻击防范上，它们并没有太大的差异。然而物联网时代的安全往往比互联网时代重要得多。

8.3.1 网络攻击

当我们使用电脑去访问某一个有问题的网站，电脑有可能会因此而感染上病毒，最坏的结果就是整个电脑完全毁坏。虽然我们并不需要主动去访问别人的物联网，但是别人可以主动访问我们的设备。如果我们在安全性上有所欠缺，这会带来极大的风险。一个骇客可能会控制我们的电灯、暖气、空调等，如果你的门是自动的，那么风险更大。

特别是，多数系统在设计的初期会考虑到一些特殊的情况，如用户忘记密码，

这时就需要去重置账户，而别人也可以在这时重置我们的账户。如果在重置的过程中，我们使用一些不安全的通信方式来接收数据，那么我们的数据就会被截获，这时我们的人身安全都将暴露在风险中。由于我并不是这方面的专家，在这里只介绍几种常见的攻击模式。

1. 密码破解

密码破解是一种很常见的攻击方式。在那些以单个用户为目的的攻击中，通常会借以社会工程学的方式来进行攻击。在以多个用户为目的攻击中，可能会使用暴力破解法，将密码进行逐个推算直到找出真正的密码为止。在用户名和密码不断泄露的今天，一些黑客可能拥有大量的已知用户和密码，这时他们就会使用**碰库攻击**的方式。因为大部分用户在不同的平台上都会使用同一个密码。黑客不需要再一个个去破解密码，只需要拿着已有的用户和密码在上面尝试登录即可。除此之外，黑客也会使用一些现有的软件的漏洞来破解密码。

2. 本地网络控制硬件

由于找不到合适的名字，暂时以这个名字来命名这种方式。当我们使用蓝牙、ZigBee 等本地的通信网络时，会遇到这样的问题——我们使用手机通过蓝牙来控制硬件，而由于蓝牙可能采用 BLE 的通信方式，如果在这个过程中，我们的通信指令被黑客知道了，那么他就可以控制硬件。笔者在写作本书的期间也破解过类似的系统：先反编译手机上的 APK，获取控制指令；接着用抓包工具拦截 UUID 码，再自己编写应用来控制硬件。尽管这只是一个很普通的设备，通过这个设备黑客可以做一些更危险的事情。

3. 拒绝服务与分布式拒绝服务

通常来说，这两种攻击方式是黑客要阻止或禁止通信设施的正常使用和管理。这两种攻击方式的目的是使被攻击的服务器不可用。换句话说，这种类型攻击的目的是让正常用户无法访问你的网站——通过不断地向网站的服务器发出请求，直到你的网站无法接受请求、资源枯竭，如带宽耗尽。因此，这种攻击多数来源于竞争

对手或者勒索等目的。

对于一般的拒绝服务攻击来说，我们可能提升服务器的配置就可以解决问题。而对于分布式拒绝服务攻击来说，就不是那么简单的事了。分布式拒绝服务是拒绝服务的升级版，它使用了许多分布的主机同时攻击一个目标。如果我们要对抗这种类型的攻击，就意味着我们在系统设计的初期就应该考虑**分布式解决方案**。除此以外我们并没有更好的选择，因为这种攻击难以从根本上杜绝。

而如果我们的服务器仅仅只对内部开放，不妨考虑使用 VPN 的解决方案，然而这很难成为一个真正的物联网应用。

4. 重放

重放指攻击者发送一个目的主机已接收过的包，来达到欺骗系统的目的。既然我们已经对其进行加密，也防止不了重放攻击。如当我们登录某个网站时，可能会生成 **http:///login?userid=ID&password=加密的密码** 这样的 URL。虽然攻击者看不到加密后的密码，但是攻击者可以用这个 URL 直接登录。这种攻击方式通常用于用户登录时，这时可以采用一些如时间戳、序列号递增、提问应答等方式来防范。

8.3.2 认证

认证又可以称为身份验证、验证、鉴权，它用来确认你所说的你确实是你。在生活中最常见的例子就是身份证，它用来标识我即是我。除此之外，还有我们脑子中老是记不住的各个网站的用户名和密码。这是一种**基于共享密钥**的身份验证，即在服务器端和用户共同拥有一个或一组密码。当我们需要登录某个网站时，我们就需要输入用户名和密码。如果用户名、密码和服务器的用户名、密码相匹配，那么这个用户就是合法用户。当我们成功登录时就会生成相对应的 Cookie，每次我们做一些相应的操作时都会在请求上带上这个 Cookie。

而在硬件上使用 Cookie 并不是一件容易的事，这时我们就需要一些别的方法。当然，每次请求时都带上用户名和密码也是一种可行的方案。在上一章中用到过“username”，然而任何人都可以使用这个“username”来访问。对于一个硬件设备来

说，我们可以在一开始的时候写定一个固定的“username”。这里的“username”只是一个简单的标识，没有任何处理。有时候黑客可以通过**猜测凭据**来模仿一个客户端，向服务端发起攻击。一种更好的方法就是通过“username”来生成一个 Token，如“phodal”经过 md5 加密后就会变成“a1eb5e02c050ae0e0bcfe8354df35fc0”。每次请求的时候，我们就带上这样的 Token，用于授权。如笔者之前实现的一个开源的物联网系统 Lan（Github: http://github.com/phodal/lan）就基于这种机制。

除此之外，还有公开密钥加密算法的身份验证。使用这种验证方式时，通信中的双方分别持有公开密钥和私有密钥。由我们的硬件端持有这个私有密钥，服务端持有公开密钥对数据进行解密，如果解密成功就认为用户是合法用户。

8.3.3 私有物联网

在云计算发展出来的时候出现了公有云和私有云，对于同样是以 Web 服务为主的物联网，既会出现公有物联网，也会出现私有物联网。私有相比于公有来说就意味着在某种程度上的安全。私有云是为一个客户单独使用而构建的，因而提供对数据、安全性和服务质量的最有效控制。私有云所属公司拥有基础设施，并可以控制在此基础设施上部署应用程序的方式。私有云可部署在企业数据中心的防火墙内，也可以部署在一个安全的主机托管场所，私有云的核心属性是专有资源。

举例来说，如果我们是一家智能电器的产商，那么由我们自己开发物联网难度显然会比较大。而选择公有物联网的话，太开放反而是一种缺点。这也是我们这个时代 SaaS 受欢迎的一个缘故，SaaS 提供的是一个软件。物联网也是一个软件，对于服务提供商来说，他们还需要考虑硬件。对于需要服务的硬件厂商来说，他们只需要考虑协议。

8.3.4 隐私

现在隐私是一个非常重要的话题。用户在使用你的产品的时候，把他/她的信息交给了你，你就应该帮用户保管好这些信息。如果你保护不了用户的隐私，为什么你还想要这么多信息呢？公司收集用户信息和数据，都出于商业目的，即使不是以

出售这些信息为目的，也会想着从数据中挖掘出一些有用的信息。数据本身不具有价值，有价值的是从数据中挖掘出来的信息。没有人想因为曾经在某个购物网站上搜索了某个产品，而在以后的所有网站的广告上看到类似产品的广告。用户与公司之间要相互依赖，你背叛了用户，那么用户就会背叛你。

对于隐私来说也是如此，作为一个开发者，在设计初期应该尽量去保证用户的隐私不会被泄露。过去我们泄露的可能是个人的一些联系方式、家庭住址。而在物联网时代，我们泄露的就会是家庭安防布局、用电器情况、是否有人在家等。甚至一些骇客可能会去调用你家的摄像头来观察你的一举一动。而这些有可能仅仅是因为你在网站上放置了一些用户的 ID。过去，我们在网络上留一个言别人就会知道我们的 IP。未来，别人通过这个 IP 可能就会知道我们的一切信息。更有甚者，有的人可以通过我们所暴露在外的接口做一些危及人身安全的事情。

8.4 小结

在最后一章里，我们对物联网的智能化及安全做了一个简单的概述和介绍。安全应该从系统设计的初期就开始考虑，而不是等到系统完成后再考虑。一个不带任何安全考虑的物联网系统，不仅可能危害用户，也有可能危害国家安全。在这一章中我们简要地介绍了一些黑客的攻击模式，由于笔者在这方面的研究有限，建议读者参考更专业的书籍。智能化则是技术人员非常感兴趣的话题，从推荐系统到自然语言处理都是很有意思的内容。

本章依旧在向读者展示我接触过的一些有意思的、可以扩展物联网相关的技术知识。相信读者可以在一章一章循序渐进的物联网系统设计中，掌握一些实用的技巧。当然，我相信这也是一本有点困难的书——物联网是一个很广阔的领域。在这个领域里，混杂着不同的计算机知识——软件、硬件，从软件上分又有前端、后台、移动应用、上位机等，从硬件上分又有不同的传感器、控制器、执行器，细分也有不同的电子元件。

最后，希望这本书可以帮你解决更多的问题。

8.5 相关阅读资料

[1] [美] Steven Bird Ewan Klein Edward LoperPython. 自然语言处理[M]. 张旭，崔阳，刘海平译. 北京：人民邮电出版社，2014.

[2] [美] Peter Harrington. 机器学习实战[M]. 李锐，李鹏，曲亚东，王斌译. 北京：人民邮电出版社，2013.

[3] [美] Toby Segaran. 集体智慧编程[M]. 莫映，王开福译. 北京：电子工业出版社，2009.

附录 A

Raspberry Pi 快速指南

Raspberry Pi 是一款针对电脑业余爱好者、教师、小学生及小型企业等用户的迷你电脑，预装 Linux 系统，体积仅信用卡大小，搭载 ARM 架构处理器，运算性能和智能手机相仿。在接口方面，Raspberry Pi 提供了可供键鼠使用的 USB 接口，此外还有千兆以太网接口、SD 卡扩展接口及 1 个 HDMI 高清视频输出接口，可与显示器或者 TV 相连。

Raspberry Pi 相比于一般的 ARM 开发板来说，由于其本身搭载着 Linux 操作系统，可以用诸如 Python、Ruby 或 Bash 来执行脚本，而不是通过编译程序来运行，具有更高的开发效率。

今天的 Raspbian 默认已经安装 openssh-server，并默认开启了 OpenSSH-Server。

接着我们就可以看到系统启动了，输入用户名和密码：

```
Raspbian GNU/Linux 7 raspberrypi ttyAMA0

raspberrypi login: pi
Password:
Last login: Sat Apr 26 05:58:07 UTC 2014 on ttyAMA0
Linux raspberrypi 3.10.25+ #622 PREEMPT Fri Jan 3 18:41:00 GMT 2014 armv6l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
```

```
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
ls

NOTICE: the software on this Raspberry Pi has not been fully configured.
Please run 'sudo raspi-config'
```

然后：

```
sudo raspi-config
```

选择第一个，下面就可以继续了：

```
Expand Filesystem          Ensures that all of the SD card s
```

重启后，便可以扩展 SD 卡成功。

注：Raspbian 与一般的 Debian 系统使用起来区别不是太大（ps：命令上），由于 CPU 是不同的架构，在编译上可能有所区别。通常 PC 上的软件需要重新编译才能在 RPi 上运行，所以如果可以用 apt-get 安装的话，就不要自己编译了。

附录 B

JavaScript 基础

今天，JavaScript 已经无处不在，也许你正打开某个网站，它的后台语言可能就是 Node.js 驱动的，或者它的 API 是 Node.js 驱动的。并且如果这个网站提供了一些很不错的交互方式，那么它必然用到了 JavaScript。虽然你还没理解上面那些是什么，也正是因为你不理解才需要去学习更多的东西。但是你要知道 JavaScript 已经无处不在了，它可能就在你手机上的某个 APP 里、在你浏览的网页里、运行在你 IDE 中的某个进程里。

B.1 hello,world

这里我们还需要有一个 helloworld.html，JavaScript 是专为网页交互而设计的脚本语言，所以我们一点点来开始这部分的旅途，先写一个符合标准的 helloworld.html。

```
<!DOCTYPE html>
<html>
    <head></head>
    <body></body>
</html>
```

然后开始融入我们的 JavaScript，向 HTML 中插入 JavaScript 的方法，就需要用到 HTML 中的<script>标签，我们先用页面嵌入的方法来写 hello, world。

```
<!DOCTYPE html>
```

```
<html>
    <head>
        <script>
            document.write('hello,world');
        </script>
    </head>
    <body></body>
</html>
```

按照标准的写法，我们还需要声明这个脚本的类型：

```
<!DOCTYPE html>
<html>
    <head>
        <script type="text/javascript">
            document.write('hello,world');
        </script>
    </head>
    <body></body>
</html>
```

没有显示 hello,world 怎么办？试试下面的代码：

```
<!DOCTYPE html>
<html>
    <head>
        <script type="text/javascript">
            document.write('hello,world');
        </script>
    </head>
    <body>
        <noscript>
            disable Javascript
        </noscript>
    </body>
</html>
```

B.2 更 JS 一点

我们需要让代码看上去更像 JS，同时是以 js 结尾。就像 C 语言的源码是以 c 结尾的，我们也同样需要让代码看上去更正式一点。于是我们需要在 helloworld.html 的同一文件夹下创建一个 app.js 文件，在里面写入：

```
document.write('hello,world');
```

同时我们的 helloworld.html 还需要告诉浏览器 JS 代码在哪里：

```
<!DOCTYPE html>
<html>
    <head>
        <script type="text/javascript" src="app.js"></script>
    </head>
    <body>
        <noscript>
            disable Javascript
        </noscript>
    </body>
</html>
```

B.3 从数学出发

让我们回到第 1 章讲述的小明的问题，从实际问题下手编程，更容易学会编程。小学时代的数学题最喜欢这样子了——某商店里的糖一个 5 块钱，小明买了 3 个糖，小明一共花了多少钱。在编程方面，也许我们还算是小学生。最直接的方法就是直接计算 3×5=?

```
document.write(3*5);
```

document.write 实际上我们也可以理解为输出，也就是往页面里写入 3*5 的结果，在有双引号的情况下会输出字符串。这时会在浏览器上看到 15，这便是一个好的开始，也是一个糟糕的开始。

B.4 设计和编程

对于实际问题，如果我们只是止于所要得到的结果，很多年之后，我们就成为了 code monkey。对这个问题进行再一次设计，所谓的设计有些时候会把简单的问题复杂化，有些时候会使以后的扩展更加简单。比如，这一天因为这家商店的糖价格太高了，于是店长将价格降为了 4 块钱。

```
document.write(3*4);
```

于是我们又得到了结果，但是下次我们看到这些代码的时候没有分清楚哪个是糖的数量，哪个是价格，于是我们重新设计了程序：

```
tang=4;
num=3;
document.write(tang*num);
```

这才能叫得上程序设计，或许你注意到了“;”这个符号的存在，我想说的是这是另外一个标准，我们不得不去遵守。

B.5 函数

记得刚开始学三角函数的时候，我们会写：

```
sin 30=0.5
```

而我们的函数也类似于此，换句话说，因为很多搞计算机的先驱都学好了数学，都把数学世界的规律带到了计算机世界，所以我们的函数也类似于此，让我们做一个简单的开始。

```
function hello(){
   return document.write("hello,world");
}
hello();
```

当我第一次看到函数的时候，有些小激动。我写了一个叫 hello 的函数，它返回了往页面中写入 hello,world 的方法，然后调用了 hello 这个函数，于是页面上有了

hello,world。

```
function sin(degree){
   return document.write(Math.sin(degree));
}
sin(30);
```

这里 degree 就称之为变量。于是输出了-0.9880316240928602，而不是 0.5，因为这里用的是弧度制，而不是角度制。

```
sin(30)
```

输出结果有点类似于 sin 30。括号是为了方便解析，这个在不同的语言中可能是不一样的，比如在 Ruby 中我们可以直接用类似于数学中的表达：

```
2.0.0-p353 :004 > Math.sin 30
=> -0.9880316240928618
2.0.0-p353 :005 >
```

我们可以在函数中传入多个变量，于是再回到小明的问题，就会这样去编写代码：

```
function calc(tang,num){
   result=tang*num;
   document.write(result);
}
calc(3,4);
```

但是从某种程度上来说，我们的 calc 既做了计算的事又做了输出的事，总的来说设计上有些不好。

B.6 重新设计

我们将输出的工作移到函数的外面：

```
function calc(tang,num){
   return tang*num;
}
document.write(calc(3,4));
```

接着我们用一种更有意思的方法来写这个问题的解决方案：

```
function calc(tang,num){
   return tang*num;
}
function printResult(tang,num){
   document.write(calc(tang,num));
}
printResult(3, 4)
```

看上去更专业了一点，如果计算的时候我们只需要调用 calc，需要输出的时候就调用 printResult 方法。

B.7 object 和函数

我们还没有说清楚之前遇到过的 document.write 以及 Math.sin 的语法为什么看上去很奇怪，让我们看看它们到底是什么，修改 app.js 为以下内容：

```
document.write(typeof document);
document.write(typeof Math);
```

typeof document 会返回 document 的数据类型，输出的结果是：

```
object object
```

所以我们需要去弄清楚什么是 object，即什么是对象。对象的定义是：无序属性的集合，其属性可以包含基本值、对象或者函数。

创建一个 object，然后观察，这便是我们接下来要做的：

```
store={};
store.tang=4;
store.num=3;
document.write(store.tang*store.num);
```

于是我们就有了和 document.write 一样的用法，这也是对象的美妙之处，只是这里的对象只包含基本值，因为：

```
typeof story.tang="number"
```

一个包含对象的对象应该是这样子的：

```
store={};
store.tang=4;
store.num=3;
document.writeln(store.tang*store.num);

var wall=new Object();
wall.store=store;
document.write(typeof wall.store);
```

而我们用到的 document.write 和上面用到的 document.writeln 都属于这个无序属性集合中的函数。

下面的代码说的就是这个无序属性集中的函数：

```
var IO=new Object();
function print(result){
   document.write(result);
};
IO.print=print;
IO.print("a obejct with function");
IO.print(typeof IO.print);
```

我们定义了一个叫 IO 的对象，声明对象可以用：

```
var store={};
```

又或者是：

```
var store=new Object{};
```

两者是等价的，但是用后者可读性会更好一点，我们定义了一个叫 print 的函数，它的作用也就是 document.write，IO 中的 print 函数等价于 print()函数，这也就是对象和函数之间的一些区别，对象可以包含函数，对象是无序属性的集合，其属性可以包含基本值、对象或者函数。

复杂一点的对象应该是下面这样的一种情况：

```
var Person={name:"phodal",weight:50,height:166};
function dream(){
```

```
    future;
};
Person.future=dream;
document.write(typeof Person);
document.write(Person.future);
```

这些在我们未来的实际编程过程中用得更多。

B.8 面向对象

开始之前先让我们简化前面的代码：

```
Person.future=function dream(){
    future;
}
```

看上去比前面的简单多了，不过我们还可以简化为下面的代码：

```
var Person=function(){
    this.name="phodal";
    this.weight=50;
    this.height=166;
    this.future=function dream(){
        return "future";
    };
};
var person=new Person();
document.write(person.name+"<br>");
document.write(typeof person+"<br>");
document.write(typeof person.future+"<br>");
document.write(person.future()+"<br>");
```

只是这时候 Person 是一个函数，但是我们声明的 person 却变成了一个对象，一个 JavaScript 函数也是一个对象，并且，所有的对象从技术上讲也只不过是函数。这里的
是 HTML 中的元素，称为 DOM，它起的是换行的作用，我们会在稍后介绍它，这里我们先关心一下 this。this 关键字表示函数的所有者或作用域，也就是这里的 Person。

上面的方法显得有点不可取，换句话说和一开始的

```
document.write(3*4);
```

一样，不具有灵活性，因此在我们完成功能之后，需要对其进行优化，这就是程序设计的真谛——解决完实际问题后，我们需要开始真正的设计，而不是解决问题时的编程。

```
var Person=function(name,weight,height){
    this.name=name;
    this.weight=weight;
    this.height=height;
    this.future=function(){
        return "future";
    };
};
var phodal=new Person("phodal",50,166);
document.write(phodal.name+"<br>");
document.write(phodal.weight+"<br>");
document.write(phodal.height+"<br>");
document.write(phodal.future()+"<br>");
```

于是，产生了这样一个可重用的 JavaScript 对象，this 关键字确立了属性的所有者。

B.9 其他

JavaScript 还有一个很强大的特性——原型继承，不过这里我们先不考虑这个，用尽量少的代码及关键字来实现所要表达的核心功能，这才是这里的核心，其他的东西我们可以从其他书本上学到。

所谓的继承：

```
var Chinese=function(){
    this.country="China";
}
```

```
var Person=function(name,weight,height){
    this.name=name;
    this.weight=weight;
    this.height=height;
    this.futrue=function(){
        return "future";
    }
}
Chinese.prototype=new Person();

var phodal=new Chinese("phodal",50,166);
document.write(phodal.country);
```

完整的 JavaScript 应该由下列三个部分组成：

- 核心（ECMAScript）——核心语言功能。
- 文档对象模型（DOM）——访问和操作网页内容的方法和接口。
- 浏览器对象模型（BOM）——与浏览器交互的方法和接口。

我们上面讲的都是 ECMAScript，也就是语法相关的，但是 JS 真正强大的，或者说我们最需要的可能就是对 DOM 的操作，这也就是为什么 jQuery 等库可以流行的原因之一，而核心语言功能才是真正在哪里都适用的，至于 BOM，真正用到的机会很少，因为没有完善的统一的标准。

一个简单的 DOM 示例：

```
<!DOCTYPE html>
<html>
<head>
</head>
<body>
    <noscript>
        disable Javascript
    </noscript>
    <p id="para" style="color:red">Red</p>
</body>
    <script type="text/javascript" src="app.js"></script>
</html>
```

我们需要修改一下 helloworld.html，添加：

```
<p id="para" style="color:red">Red</p>
```

同时还需要将 script 标签移到 body 下面，如果没有意外的话，我们会看到页面上用红色的字体显示 Red，修改 app.js：

```
var para=document.getElementById("para");
para.style.color="blue";
```

接着，字体就变成了蓝色，有了 DOM 我们就可以对页面进行操作，可以说我们看到的绝大部分页面效果都是通过 DOM 操作实现的。

附录 C

Ionic 简单帮助文档

C.1 Ionic 路由过程

负责 Ionic 路由的是 Angular UI Router。它是 AngularJS 的路由框架，和 Angular 默认的$route 不同，它将所有路由包装成可划分层级的状态机状态，路由路径在 ui-router 中不是必需的。由于 ui-router 的路由状态机是分层级的，所以使用 ui-router 可以非常方便地创建包含多个嵌入的子模板。

一个简单的路由如下所示：

```
.state('app', {
    url: "/app",
    abstract: true,
    templateUrl: "templates/menu.html"
})
```

即有 URL、模板，而对于一个复杂的 Ionic 的 Router 来说是这样子的：

```
.state('app.wiki', {
    url: "/wiki",
    views: {
        'menuContent': {
            templateUrl: "templates/wiki.html",
```

```
            controller: 'WikiCtrl'
        }
    }
})
```

即带有 Controller，一个完整的 Router 还会有一个默认的路由：

```
$urlRouterProvider.otherwise('/app/level');
```

C.2 路由

/app/level 对应的路由是：

```
.state('app.levelSelect', {
  url: "/level",
  views: {
    'menuContent': {
      templateUrl: "templates/level.html",
      controller: 'LevelSelectCtrl'
    }
  }
})
```

level.html 的内容如下所示：

```
<ion-view view-title="选择级别">
  <ion-content>
    <ion-list>
      <ion-item ng-repeat="level in levels" href="#/app/level/{{level.id}}">
          <div ng-if="level.id == 1 ">
              <i class="icon ion-ios-unlocked"></i>  {{level.title}}
          </div>
          <div ng-if="level.id != 1 ">
              <i class="icon ion-ios-locked"></i>  {{level.title}}
          </div>
      </ion-item>
```

```
    </ion-list>
  </ion-content>
</ion-view>
```

而 LevelSelectCtrl 如下所示：

```
.controller('LevelSelectCtrl', function ($scope) {
    if (typeof analytics !== 'undefined') {
        analytics.trackView("LevelSelectCtrl");
    }
    $scope.levels = [
        {title: 'Level 1', id: 1},
        {title: 'Level 2', id: 2},
        {title: 'Level 3', id: 3},
        {title: 'Level 4', id: 4}
    ];
})
```

Controller 做的事似乎就是将数据放到模板，接着渲染，所以就会有如下的结果：

[ScreentShot][1]

当我们选择 level1 时，逻辑就来到了：

```
.state('app.single', {
  url: "/level/:level",
  views: {
    'menuContent': {
      templateUrl: "templates/level_quiz.html",
      controller: 'QuizCtrl'
    }
  }
})
```

这就是一个简单的页面间的跳转与 Router 的关系。

C.3 发布

1. 发布 iOS 应用和 Windows Phone 应用

由于发布 iOS 应用还需要依赖于 XCode，而直接使用 Ionic 则只能生成相应的源码。同理还有 Windows Phone 应用，依赖于 Visual Studio，也生成相应的源码。

剩下的步骤则需要参考相应的官方文档。

2. 发布 Android 应用

在 Release 之前，你需要了解两件事情：

（1）Keytool 是一个有效的安全钥匙和证书的管理工具。

（2）Android 要求所有的程序必须有签名，否则就不会安装该程序。

生成命令（注意：记得将 alias_name 改一下）：

```
$ keytool -genkey -v -keystore my-release-key.keystore -alias alias_name
-keyalg RSA -keysize 2048 -validity 10000
```

之前忘记修改这个名字造成很大问题，这过程中会问你一些问题：

```
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  phodal
What is the name of your organizational unit?
  [Unknown]:  phodal
What is the name of your organization?
  [Unknown]:  phodal
What is the name of your City or Locality?
```

注意：如果你想将应用发布到一些应用市场，则请保存好你的签名，如果出现签名不一致的话，就说明有问题了。

接着就可以依据下面的步骤生成软件包了。

（1）生成 release 包

Cordova 提供了一组设备相关的 API，通过这组 API，移动应用能够以 JavaScript 访问原生的设备功能，如摄像头、麦克风等。

```
$ cordova build --release android
```

（2）签名

jarsigner 是 JDK 中包含的用于 JAR 文件签名和验证的工具。

```
$jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore
my-release- key.keystore HelloWorld-release-unsigned.apk alias_name
```

注意：alias_name 与上面的应该是一致的。

（3）优化

Zipalign 是一个 Android 平台上整理 APK 文件的工具，它首次被引入是在 Android 1.6 版本的 SDK 软件开发工具包中。它能够对打包的 Android 应用程序进行优化，以使 Android 操作系统与应用程序之间的交互更有效率，这能够让应用程序和整个系统运行得更快。

```
$zipalign -v 4 HelloWorld-release-unsigned.apk HelloWorld.apk
```

注意：正常情况下，zipalign 应该在你的 Android SDK 的 Home 目录的 build-tools 的某个版本的 SDK 下面。

附录 D

相关资源

XMPP 是一种基于标准通用标记语言的子集 XML 的协议，它继承了在 XML 环境中灵活的发展性。因此，基于 XMPP 的应用具有超强的可扩展性。经过扩展以后的 XMPP 可以通过发送扩展的信息来处理用户的需求，以及在 XMPP 的顶端建立如内容发布系统和基于地址的服务等应用程序。而且，XMPP 包含了针对服务器端的软件协议，使之能够与另一个服务器进行通话，这使得开发者更容易建立客户应用程序或给一个配置好的系统添加功能。

Thread 是一种基于简化版 IPv6 的网状网络协议，该协议由行业领先的多家技术公司联合开发，旨在实现家庭中各种产品间的互联，以及与互联网和云的连接。Thread 基于低成本、低功耗的 802.15.4 芯片组开发，Thread 易于安装、高度安全，并且可扩展到数百台设备。目前正在使用的大量产品，只需一次简单的软件升级，便可支持 Thread。

Z-Wave 是一种新兴的基于射频的、低成本、低功耗、高可靠、适于网络的短距离无线通信技术。工作频带为 908.42MHz（美国）~868.42MHz（欧洲），采用 FSK（BFSK/GFSK）调制方式，数据传输速率为 9.6 kbps，信号的有效覆盖范围在室内是 30m，室外可超过 100m，适合于窄带宽应用场合。随着通信距离的增大，设备的复杂度、功耗以及系统成本都在增加，相对于现有的各种无线通信技术，Z-Wave 技术将是最低功耗和最低成本的技术，有力地推动着低速率无线个人区域网。

ZigBee 是基于 IEEE 802.15.4 标准的低功耗局域网协议。根据国际标准规定，

ZigBee 技术是一种短距离、低功耗的无线通信技术。这一名称（又称紫蜂协议）来源于蜜蜂的八字舞，由于蜜蜂（bee）靠飞翔和“嗡嗡”（zig）地抖动翅膀的“舞蹈”来与同伴传递花粉所在的方位信息，也就是说，蜜蜂依靠这样的方式构成了群体中的通信网络。其特点是近距离、低复杂度、自组织、低功耗、低数据速率。主要适合用于自动控制和远程控制领域，可以嵌入各种设备。简而言之，ZigBee 就是一种便宜的、低功耗的近距离无线组网通信技术。

6LoWPAN 取得的突破是得到一种非常紧凑、高效的 IP 实现，消除了以前造成的各种专门标准和专有协议的因素。这在工业协议（BACNet、LonWorks、通用工业协议和监控与数据采集）领域具有特别的价值。这些协议最初的开发是为了提供特殊的行业特有的总线和链路（从控制器区域网总线到 AC 电源线）上的互操作性。

uIP 由瑞典计算机科学学院（网络嵌入式系统小组）的 Adam Dunkels 开发。其源代码由 C 语言编写，并且完全公开。uIP 协议栈去掉了完整的 TCP/IP 中不常用的功能，简化了通信流程，但保留了网络通信必须使用的协议，设计重点放在了 IP/TCP/ICMP/UDP/ARP 这些网络层和传输层协议上，保证了其代码的通用性和结构的稳定性。

NFC 近场通信技术是由非接触式射频识别（RFID）及互联互通技术整合演变而来，在单一芯片上结合感应式读卡器、感应式卡片和点对点的功能，能在短距离内与兼容设备进行识别和数据交换。工作频率为 13.56MHz。但是使用这种手机支付方案的用户必须更换特制的手机。目前这项技术在日韩被广泛应用。手机用户凭着配置了支付功能的手机就可以行遍全国：他们的手机可以用作机场登机验证、大厦的门禁钥匙、交通一卡通、信用卡、支付卡等。

WiFi 是一种可以将个人电脑、手持设备（如 Pad、手机）等终端以无线方式互相连接的技术，事实上它是一个高频无线电信号。无线保真是一个无线网络通信技术的品牌，由 Wi-Fi 联盟所持有，目的是改善基于 IEEE 802.11 标准的无线网路产品之间的互通性。有人把使用 IEEE 802.11 系列协议的局域网称为无线保真，甚至把无线保真等同于无线网际网路（Wi-Fi 是 WLAN 的重要组成部分）。

FreeRTOS 是一个迷你操作系统内核的小型嵌入式系统。作为一个轻量级的操作

系统，功能包括：任务管理、时间管理、信号量、消息队列、内存管理、记录功能等，可基本满足较小系统的需要。由于 RTOS 需占用一定的系统资源（尤其是 RAM 资源），只有 μC/OS-II、embOS、salvo、FreeRTOS 等少数实时操作系统能在小 RAM 单片机上运行。相对于 μC/OS-II、embOS 等商业操作系统，FreeRTOS 操作系统是完全免费的操作系统，具有源码公开、可移植、可裁减、调度策略灵活的特点，可以方便地移植到各种单片机上运行，其最新版本为 8.0.0 版。

LwIP 是 Light Weight（轻型）IP 协议，有无操作系统的支持都可以运行。LwIP 实现的重点是在保持 TCP 协议主要功能的基础上减少对 RAM 的占用，它只需十几 KB 的 RAM 和 40KB 左右的 ROM 就可以运行，这使得 LwIP 协议栈适合在低端的嵌入式系统中使用。lwIP 协议栈主要关注的是如何减少内存的使用和代码的大小，这样就可以让 lwIP 适用于资源有限的小型平台，如嵌入式系统。为了简化处理过程和内存要求，lwIP 对 API 进行了裁减，可以不需要复制一些数据。

Contiki 是一个适用于有内存的嵌入式系统的开源的、高可移植的、支持网络的多任务操作系统。包括一个多任务核心、TCP/IP 堆栈、程序集及低能耗的无线通信堆栈。Contiki 是采用 C 语言开发的非常小型的嵌入式操作系统，运行时只需要几 KB 的内存。Contiki 是一个小型的、开源的、极易移植的多任务电脑操作系统。它专门设计以适用于一系列的内存受限的网络系统，包括从 8 位电脑到微型控制器的嵌入系统。它的名字来自于托尔·海尔达尔的康提基号。Contiki 只需几 kilobyte 的代码和几百字节的内存就能提供多任务环境和内建 TCP/IP 支持。

embOS 是一个优先级控制的多任务系统，是专门为各种微控制器应用于实时系统应用的嵌入式操作系统，是一个具有最小 RAM 和 ROM 占用的、高速的、多功能的高性能工具。贯穿 embOS 的整个开发过程，微控制器有限的资源一直是开发者所顾忌的。5 年来，该 RTOS 的内部结构已经被优化到不同客户的不同应用中，以满足工业需要。对不同微控制器的完全源码，使开发者可以很方便地用实时操作系统构建实时程序。embOS 是高度模块化的，只有需要的函数才被调用，占用的 ROM 非常小。最小的内存占用：1KB ROM，30 字节 RAM。由于提供源码文件，你可以用 embOS 灵活定制系统以满足实际需求。 任务之间可以通过旗语、邮箱和事件安全便

利地通信。

uC/OS II（Micro Control Operation System Two）是一个可以基于 ROM 运行的、可裁减的、抢占式、实时多任务内核，具有高度可移植性，特别适合于微处理器和控制器，是和很多商业操作系统性能相当的实时操作系统（RTOS）。为了提供最好的移植性能，uC/OS II 最大程度上使用 ANSI C 语言进行开发，并且已经移植到近四十多种处理器体系上，涵盖了从 8 位到 64 位各种 CPU（包括 DSP）。uC/OS II 可以简单地视为一个多任务调度器，在这个任务调度器之上完善并添加了和多任务操作系统相关的系统服务，如信号量、邮箱等。其主要特点有公开源代码、代码结构清晰明了、注释详尽、组织有条理、可移植性好、可裁剪、可固化。内核属于抢占式，最多可以管理 60 个任务。从 1992 年开始，由于其高度可靠性、移植性和安全性，uC/OS II 已经广泛使用在从照相机到航空电子产品的各种应用中。

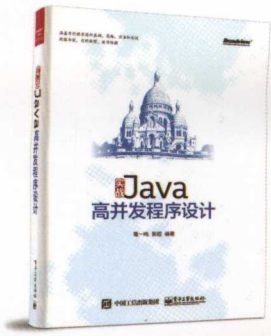
TinyOS 是 UC Berkeley（加州大学伯克利分校）开发的开放源代码操作系统，专为嵌入式无线传感网络设计，操作系统基于构件（component-based）的架构使得快速更新成为可能，而这又减小了受传感网络存储器限制的代码长度。

TinyOS 的构件包括网络协议、分布式服务器、传感器驱动及数据识别工具。其良好的电源管理源于事件驱动执行模型，该模型也允许时序安排，具有很好灵活性。TinyOS 已被应用于多个平台和感应板中。

OpenWrt 可以被描述为一个嵌入式的 Linux 发行版，（主流路由器固件有 dd-wrt、tomato、openwrt 三类）而不是试图建立一个单一的、静态的系统。OpenWrt 的包管理提供了一个完全可写的文件系统，从应用程序供应商提供的选择和配置，并允许自定义设备，以适应任何应用程序。对开发人员来说，OpenWrt 使用框架来构建应用程序，而无须建立一个完整的固件来支持。对用户来说，这意味着其拥有完全定制的能力，可以用前所未有的方式使用该设备。

QNX 是由加拿大 QSSL 公司（QNX Software System Ltd.）开发的分布式实时操作系统。该操作系统既能运行于以 Intel X86、Pentium 等 CPU 为核心的硬件环境，也能运行于以 PowerPC、MIPS 等 CPU 为核心的硬件环境。QNX 操作系统符合 POSIX 基本标准和实时标准，使其应用可以方便地进行移植。

好书分享



欢迎反馈意见或投稿
邮箱：dongying@phei.com.cn
电话：010-88254047
微信号：yingzidd

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

物联网是一个跨学科的新兴领域。物联网应用系统的设计和实现，涉及门类众多的硬件、软件和网络通信知识。无论是技术选型还是具体的实现，设计者可能都需要面对各种各样的挑战和困难。作者运用目前主流的技术，用清晰流畅的表述方式，向读者呈现了物联网系统设计和实现的实际过程。作者是GitHub上的活跃开发者，是物联网领域的“全栈”工程师，他的设计和实现思路值得本书大多数的读者去学习和借鉴。

——张崇明，《物联网设计：从原型到产品》译者

在如今物联网大热的环境下，很多 APP 工程师不懂硬件，很多硬件工程师不懂云，很多云工程师不懂APP。物联网是不同技术高速公路的交汇点，这本书充当了立交桥的作用，将不同的领域连通，以实战为主，由浅入深，帮助物联网开发者快速学习物联网这个交叉领域，是一本不错的好书。

——黄锐，NodeMCU 开源项目创始人

近年来物联网的概念逐渐走进人们的视野，基于物联网的应用正在改变着我们的生活，越来越多的人们期待着它在未来的广阔应用前景。这本书给我们打开了物联网的大门，既从整体上呈现了物联网的框架，又深入地剖析了物联网所涉及的各种技术细节，如多种开发板的应用实现和物联网数据传输协议MQTT和CoAP的介绍等。更加难得的是，贯穿整本书的系统架构设计，展现了实际物联网系统的设计和实现思路，使得这本书非常具备实战价值。

——张龙，绿米联创高级软件工程师

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

责任编辑：董　英
封面设计：吴海燕

上架建议：计算机>物联网

ISBN 978-7-121-29053-4

定价：59.00元